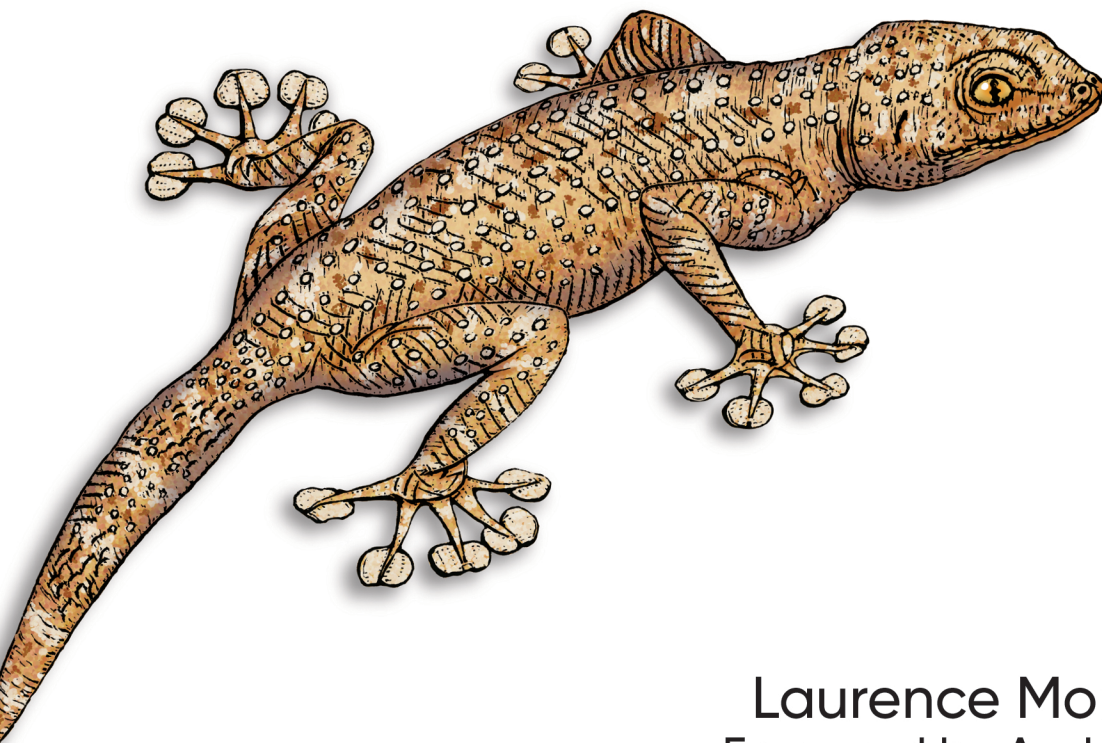


O'REILLY®

AI and Machine Learning for Coders

A Programmer's Guide to Artificial Intelligence



Laurence Moroney
Foreword by Andrew Ng

VISIT...

LANZAROTE
Caliente.COM

AI and Machine Learning for Coders

If you're looking to make a career move from programmer to AI specialist, this is the ideal place to start. Based on Laurence Moroney's extremely successful AI courses, this introductory book provides a hands-on, code-first approach to help you build confidence while you learn key topics. All you need is experience with Python and its notation for data and array processing.

You'll learn how to implement the most common scenarios in machine learning, including computer vision, natural language processing (NLP), and sequence modeling for web, mobile, cloud, and embedded runtimes. Most books on machine learning begin with a daunting amount of advanced math. This guide provides practical lessons that let you work directly with the code.

- Understand machine learning basics by working with code samples
- Use TensorFlow to build models for a variety of scenarios
- Build a model with a neural network containing only one neuron
- Implement computer vision, including feature detection in images
- Tokenize and sequence words and sentences with NLP
- Embed your models in Android and iOS devices
- Serve models over the web and in the cloud with TensorFlow Serving

Laurence Moroney leads AI advocacy at Google, teaching software developers how to build AI systems with machine learning. He's a frequent contributor to the TensorFlow YouTube channel, a recognized global keynote speaker, and a prolific author.

"The book is a great introduction to understand and practice machine learning and artificial intelligence models by using TensorFlow.

—Jialin Huang PhD
Data and Applied Scientist, Microsoft

"Laurence Moroney has been a major force in building TensorFlow into one of the world's leading AI frameworks. I was privileged to support his teaching TensorFlow with deeplearning.ai and Coursera. I wish you the best in your journey learning TensorFlow. With Laurence as a teacher, great adventures await you."

—Andrew Ng
Founder, deeplearning.ai

AI

US \$59.99

CAN \$79.99

ISBN: 978-1-492-07819-7



Twitter: @oreillymedia
facebook.com/oreilly

Praise for *AI and Machine Learning for Coders*

“Machine learning should be in the toolbox of every great engineer in this coming decade. For people looking to get started, *AI and Machine Learning for Coders* by Laurence Moroney is the much-needed practical starting point to dive deep into deep learning, computer vision, and NLP.”

—Dominic Monn, *Machine Learning at Doist*

“The book is a great introduction to understand and practice machine learning and artificial intelligence models by using TensorFlow. It covers various deep learning models, and their practical applications, as well as how to utilize TensorFlow framework to develop and deploy ML/AI applications across platforms. I recommend it for anyone who is interested in ML and AI practice.”

—Jialin Huang PhD, *Data and Applied Scientist at Microsoft*

“Laurence’s book helped me refresh TensorFlow framework and Coursera Specialization, and motivated me to take the certification provided by Google. If you have time and you are willing to embark to an ML journey, this book is a starting point from the practice side.”

—Laura Uzcátegui, *Software Engineer*

“This book is a must-read for developers who would like to get into AI/ML. You will learn a variety of examples by coding instead of math equations.”

—Margaret Maynard-Reid, *ML Google Developer Expert*

“A practical handbook to have on your desk for implementing deep learning models.”

—Pin-Yu Chen, *Research Staff Member at IBM Research AI*

“A fun book to read and practice coding for AI and machine learning projects. Intuitive wording and graphs to explain the nonintuitive concepts and algorithms. Cool coding examples to teach you key building blocks for AI and ML. In the end, you can code AI projects for your PC program, Android, iOS and Browser!”

—*Su Fu, CEO of Alchemist*

AI and Machine Learning for Coders

A Programmer's Guide to Artificial Intelligence

Laurence Moroney

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

AI and Machine Learning for Coders

by Laurence Moroney

Copyright © 2021 Laurence Moroney. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Rebecca Novack

Development Editor: Angela Rufino

Production Editor: Katherine Tozer

Copyeditor: Rachel Head

Proofreader: Piper Editorial, LLC

Indexer: Judith McConville

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: O'Reilly Media, Inc.

October 2020: First Edition

Revision History for the First Edition

2020-10-01: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492078197> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *AI and Machine Learning for Coders*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-07819-7

[LSI]

Table of Contents

Foreword.....	xiii
---------------	------

Preface.....	xv
--------------	----

Part I. Building Models

1. Introduction to TensorFlow.....	1
What Is Machine Learning?	1
Limitations of Traditional Programming	3
From Programming to Learning	5
What Is TensorFlow?	7
Using TensorFlow	8
Installing TensorFlow in Python	9
Using TensorFlow in PyCharm	9
Using TensorFlow in Google Colab	12
Getting Started with Machine Learning	13
Seeing What the Network Learned	18
Summary	19
2. Introduction to Computer Vision.....	21
Recognizing Clothing Items	21
The Data: Fashion MNIST	22
Neurons for Vision	23
Designing the Neural Network	25
The Complete Code	26
Training the Neural Network	29

Exploring the Model Output	29
Training for Longer—Discovering Overfitting	30
Stopping Training	30
Summary	32
3. Going Beyond the Basics: Detecting Features in Images.	33
Convolutions	34
Pooling	35
Implementing Convolutional Neural Networks	37
Exploring the Convolutional Network	39
Building a CNN to Distinguish Between Horses and Humans	42
The Horses or Humans Dataset	42
The Keras ImageDataGenerator	43
CNN Architecture for Horses or Humans	45
Adding Validation to the Horses or Humans Dataset	47
Testing Horse or Human Images	49
Image Augmentation	52
Transfer Learning	55
Multiclass Classification	59
Dropout Regularization	63
Summary	65
4. Using Public Datasets with TensorFlow Datasets.	67
Getting Started with TFDS	68
Using TFDS with Keras Models	70
Loading Specific Versions	73
Using Mapping Functions for Augmentation	73
Using TensorFlow Addons	74
Using Custom Splits	74
Understanding TFRecord	76
The ETL Process for Managing Data in TensorFlow	78
Optimizing the Load Phase	80
Parallelizing ETL to Improve Training Performance	81
Summary	83
5. Introduction to Natural Language Processing.	85
Encoding Language into Numbers	85
Getting Started with Tokenization	86
Turning Sentences into Sequences	87
Removing Stopwords and Cleaning Text	91
Working with Real Data Sources	93

Getting Text from TensorFlow Datasets	93
Getting Text from CSV Files	97
Getting Text from JSON Files	99
Summary	101
6. Making Sentiment Programmable Using Embeddings.	103
Establishing Meaning from Words	103
A Simple Example: Positives and Negatives	104
Going a Little Deeper: Vectors	105
Embeddings in TensorFlow	106
Building a Sarcasm Detector Using Embeddings	106
Reducing Overfitting in Language Models	109
Using the Model to Classify a Sentence	119
Visualizing the Embeddings	120
Using Pretrained Embeddings from TensorFlow Hub	123
Summary	125
7. Recurrent Neural Networks for Natural Language Processing.	127
The Basis of Recurrence	127
Extending Recurrence for Language	130
Creating a Text Classifier with RNNs	132
Stacking LSTMs	134
Using Pretrained Embeddings with RNNs	139
Summary	146
8. Using TensorFlow to Create Text.	147
Turning Sequences into Input Sequences	148
Creating the Model	152
Generating Text	154
Predicting the Next Word	154
Compounding Predictions to Generate Text	155
Extending the Dataset	156
Changing the Model Architecture	157
Improving the Data	158
Character-Based Encoding	161
Summary	162
9. Understanding Sequence and Time Series Data.	163
Common Attributes of Time Series	165
Trend	165
Seasonality	165

Autocorrelation	166
Noise	166
Techniques for Predicting Time Series	167
Naive Prediction to Create a Baseline	167
Measuring Prediction Accuracy	170
Less Naive: Using Moving Average for Prediction	170
Improving the Moving Average Analysis	171
Summary	172
10. Creating ML Models to Predict Sequences.....	173
Creating a Windowed Dataset	174
Creating a Windowed Version of the Time Series Dataset	176
Creating and Training a DNN to Fit the Sequence Data	178
Evaluating the Results of the DNN	179
Exploring the Overall Prediction	181
Tuning the Learning Rate	183
Exploring Hyperparameter Tuning with Keras Tuner	185
Summary	189
11. Using Convolutional and Recurrent Methods for Sequence Models.....	191
Convolutions for Sequence Data	191
Coding Convolutions	192
Experimenting with the Conv1D Hyperparameters	195
Using NASA Weather Data	198
Reading GISS Data in Python	199
Using RNNs for Sequence Modeling	200
Exploring a Larger Dataset	203
Using Other Recurrent Methods	205
Using Dropout	206
Using Bidirectional RNNs	209
Summary	211
<hr/>	
Part II. Using Models	
12. An Introduction to TensorFlow Lite.....	215
What Is TensorFlow Lite?	215
Walkthrough: Creating and Converting a Model to TensorFlow Lite	217
Step 1. Save the Model	218
Step 2. Convert and Save the Model	219
Step 3. Load the TFLite Model and Allocate Tensors	219

Step 4. Perform the Prediction	220
Walkthrough: Transfer Learning an Image Classifier and Converting to TensorFlow Lite	222
Step 1. Build and Save the Model	222
Step 2. Convert the Model to TensorFlow Lite	223
Step 3. Optimize the Model	225
Summary	227
13. Using TensorFlow Lite in Android Apps.....	229
What Is Android Studio?	229
Creating Your First TensorFlow Lite Android App	230
Step 1. Create a New Android Project	230
Step 2. Edit Your Layout File	232
Step 3. Add the TensorFlow Lite Dependencies	234
Step 4. Add Your TensorFlow Lite Model	236
Step 5. Write the Activity Code to Use TensorFlow Lite for Inference	236
Moving Beyond “Hello World”—Processing Images	239
TensorFlow Lite Sample Apps	242
Summary	244
14. Using TensorFlow Lite in iOS Apps.....	245
Creating Your First TensorFlow Lite App with Xcode	245
Step 1. Create a Basic iOS App	245
Step 2. Add TensorFlow Lite to Your Project	247
Step 3. Create the User Interface	248
Step 4. Add and Initialize the Model Inference Class	250
Step 5. Perform the Inference	253
Step 6. Add the Model to Your App	254
Step 7. Add the UI Logic	256
Moving Beyond “Hello World”—Processing Images	258
TensorFlow Lite Sample Apps	261
Summary	261
15. An Introduction to TensorFlow.js.....	263
What Is TensorFlow.js?	263
Installing and Using the Brackets IDE	265
Building Your First TensorFlow.js Model	266
Creating an Iris Classifier	270
Summary	274

16. Coding Techniques for Computer Vision in TensorFlow.js.....	275
JavaScript Considerations for TensorFlow Developers	276
Building a CNN in JavaScript	277
Using Callbacks for Visualization	279
Training with the MNIST Dataset	282
Running Inference on Images in TensorFlow.js	288
Summary	288
17. Reusing and Converting Python Models to JavaScript.....	291
Converting Python-Based Models to JavaScript	291
Using the Converted Models	293
Using Preconverted JavaScript Models	295
Using the Toxicity Text Classifier	295
Using MobileNet for Image Classification in the Browser	298
Using PoseNet	301
Summary	304
18. Transfer Learning in JavaScript.....	305
Transfer Learning from MobileNet	306
Step 1. Download MobileNet and Identify the Layers to Use	306
Step 2. Create Your Own Model Architecture with the Outputs from MobileNet as Its Input	308
Step 3. Gather and Format the Data	310
Step 4. Train the Model	316
Step 5. Run Inference with the Model	317
Transfer Learning from TensorFlow Hub	319
Using Models from TensorFlow.org	322
Summary	324
19. Deployment with TensorFlow Serving.....	327
What Is TensorFlow Serving?	327
Installing TensorFlow Serving	330
Installing Using Docker	330
Installing Directly on Linux	331
Building and Serving a Model	332
Exploring Server Configuration	335
Summary	338
20. AI Ethics, Fairness, and Privacy.....	339
Fairness in Programming	340
Fairness in Machine Learning	343

Tools for Fairness	345
The What-If Tool	345
Facets	346
Federated Learning	349
Step 1. Identify Available Devices for Training	349
Step 2. Identify Suitable Available Devices for Training	350
Step 3. Deploy a Trainable Model to Your Training Set	350
Step 4. Return the Results of the Training to the Server	351
Step 5. Deploy the New Master Model to the Clients	351
Secure Aggregation with Federated Learning	352
Federated Learning with TensorFlow Federated	353
Google’s AI Principles	354
Summary	355
Index.....	357

Foreword

Dear Reader,

AI is poised to transform every industry, but almost every AI application needs to be customized for its particular use. A system for reading medical records is different from one for finding defects in a factory, which is different from a product recommendation engine. For AI to reach its full potential, engineers need tools that can help them adapt the amazing capabilities available to the millions of concrete problems we wish to solve.

When I led the Google Brain team, we started to build the C++ precursor to TensorFlow called DistBelief. We were excited about the potential of harnessing thousands of CPUs to train a neural network (for instance, using 16,000 CPUs to train a cat detector on unlabeled YouTube videos). How far deep learning has come since then! What was once cutting-edge can now be done for around \$3,000 of cloud computing credits, and Google routinely trains neural networks using TPUs and GPUs at a scale that was unimaginable just years ago.

TensorFlow, too, has come a long way. It is far more usable than what we had in the early days, and has rich features ranging from modeling, to using pretrained models, to deploying on low-compute edge devices. It is today empowering hundreds of thousands of developers to build their own deep learning models.

Laurence Moroney, as Google's lead AI Advocate, has been a major force in building TensorFlow into one of the world's leading AI frameworks. I was privileged to support his teaching TensorFlow with [deeplearning.ai](https://www.deeplearning.ai) and Coursera. These courses have reached over 80,000 learners and received numerous glowing reviews.

One unexpected aspect of friendship with Laurence is that he is also a free source of Irish poetry. He once Slacked me:

Andrew sang a sad old song
fainted through miss milliner
invitation hoops
fainted fainted

[...]

He had trained an LSTM on lyrics of traditional Irish songs and it generated these lines. If AI opens the door to fun like that, how could anyone not want to get involved? You can (i) work on exciting projects that move humanity forward, (ii) advance your career, and (iii) get free Irish poetry.

I wish you the best in your journey learning TensorFlow. With Laurence as a teacher, great adventures await you.

Keep learning,

— Andrew Ng
Founder, deeplearning.ai

Preface

Welcome to *AI and Machine Learning for Coders*, a book that I've been wanting to write for many years but that has only really become possible due to recent advances in machine learning (ML) and, in particular, TensorFlow. The goal of this book is to prepare you, as a coder, for many of the scenarios that you can address with machine learning, with the aim of equipping you to be an ML and AI developer *without* needing a PhD! I hope that you'll find it useful, and that it will empower you with the confidence to get started on this wonderful and rewarding journey.

Who Should Read This Book

If you're interested in AI and ML, and you want to get up and running quickly with building models that learn from data, this book is for you. If you're interested in getting started with common AI and ML concepts—computer vision, natural language processing, sequence modeling, and more—and want to see how neural networks can be trained to solve problems in these spaces, I think you'll enjoy this book. And if you have models that you've trained and want to get them into the hands of users on mobile, in the browser, or via the cloud, then this book is also for you!

Most of all, if you've been put off entering this valuable area of computer science because of perceived difficulty, in particular believing that you'll need to dust off your old calculus books, then fear not: this book takes a code-first approach that shows you just how easy it is to get started in the world of machine learning and artificial intelligence using Python and TensorFlow.

Why I Wrote This Book

I first got seriously involved with artificial intelligence in the spring of 1992. A freshly minted physics graduate living in London in the midst of a terrible recession, I had been unemployed for six months. The British government started a program to train 20 people in AI technology, and put out a call for applicants. I was the first

participant selected. Three months later, the program failed miserably, because while there was plenty of *theoretical* work that could be done with AI, there was no easy way to do it *practically*. One could write simple inference in a language called Prolog, and perform list processing in a language called Lisp, but there was no clear path to deploying them in industry. The famous “AI winter” followed.

Then, in 2016, while I was working at Google on a product called Firebase, the company offered machine learning training to all engineers. I sat in a room with a number of other people and listened to lectures about calculus and gradient descent. I couldn’t quite match this to a practical implementation of ML, and I was suddenly transported back to 1992. I brought feedback about this, and about how we should educate people in ML, to the TensorFlow team—and they hired me in 2017. With the release of TensorFlow 2.0 in 2018, and in particular the emphasis on high-level APIs that made it easy for developers to get started, I realized the need was there for a book that took advantage of this, and widened access to ML so that it wasn’t just for mathematicians or PhDs anymore.

I believe that more people using this technology and deploying it to end users will lead to an explosion in AI and ML that will prevent another AI winter, and change the world very much for the better. I’m already seeing the impact of this, from the work done by Google on diabetic retinopathy, through Penn State University and PlantVillage building an ML model for mobile that helps farmers diagnose cassava disease, Médecins Sans Frontières using TensorFlow models to help diagnose antibiotic resistance, and much, much more!

Navigating This Book

The book is written in two main parts. **Part I** (Chapters 1–11) talks about how to use TensorFlow to build machine learning models for a variety of scenarios. It takes you from first principles—building a model with a neural network containing only one neuron—through computer vision, natural language processing, and sequence modeling. **Part II** (Chapters 12–20) then walks you through scenarios for putting your models in people’s hands on Android and iOS, in browsers with JavaScript, and serving via the cloud. Most chapters are standalone, so you can drop in and learn something new, or, of course, you could just read the book cover to cover.

Technology You Need to Understand

The goal of the first half of the book is to help you learn how to use TensorFlow to build models with a variety of architectures. The only real prerequisite to this is understanding Python, and in particular Python notation for data and array processing. You might also want to explore Numpy, a Python library for numeric calculations. If you have no familiarity with these, they are quite easy to learn, and

you can probably pick up what you need as you go along (although some of the array notation might be a bit hard to grasp).

For the second half of the book, I generally will not teach the languages that are shown, but instead show how TensorFlow models can be used in them. So, for example, in the Android chapter ([Chapter 13](#)) you'll explore building apps in Kotlin with Android studio, and in the iOS chapter ([Chapter 14](#)) you'll explore building apps in Swift with Xcode. I won't be teaching the syntax of these languages, so if you aren't familiar with them, you may need a primer—*Learning Swift* by Jonathan Manning, Paris Buttfield-Addison, and Tim Nugent (O'Reilly) is a great choice.

Online Resources

A variety of online resources are used by, and supported in, this book. At the very least I would recommend that you keep an eye on [TensorFlow](#) and its associated [YouTube channel](#) for any updates and breaking changes to technologies discussed in the book.

The code for this book is available at <https://github.com/lmoroney/tfbook>, and I will keep it up to date there as the platform evolves.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, data types, environment variables, statements, and keywords.

Constant width bold

Used for emphasis in code snippets.



This element signifies a note.

Using Code Examples

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*AI and Machine Learning for Coders*, by Laurence Moroney. Copyright 2021 Laurence Moroney, 978-1-492-07819-7.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning



For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <http://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://oreil.ly/ai-ml>.

Email bookquestions@oreilly.com to comment or ask technical questions about this book.

For news and information about our books and courses, visit <http://oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

I'd like to thank lots of people who have helped in the creation of this book.

Jeff Dean, who gave me the opportunity to be part of the TensorFlow team, beginning the second phase of my AI journey. There's also the rest of the team, and while there are too many to name, I'd like to call out Sarah Sirajuddin, Megan Kacholia, Martin Wicke, and Francois Chollet for their amazing leadership and engineering!

The developer relations team for TensorFlow, led by Kemal El Moujahid, Magnus Hyttsten, and Wolff Dobson, who create the platform for people to learn AI and ML with TensorFlow.

Andrew Ng, who, as well as writing the Foreword for this book, also believed in my approach to teaching TensorFlow, and with whom I created three specializations at Coursera, teaching hundreds of thousands of people how to succeed with machine learning and AI. Andrew also leads a team at deeplearning.ai who were terrific at helping me be a better machine learner, including Ortal Arel, Eddy Shu, and Ryan Keenan.

The team at O'Reilly that made this book possible: Rebecca Novack and Angela Rufino, without whose hard work I never would have gotten it done!

The amazing tech review team: Jialin Huang, Laura Uzcátegui, Lucy Wong, Margaret Maynard-Reid, Su Fu, Darren Richardson, Dominic Monn, and Pin-Yu.

And of course, most important of all (even more than Jeff and Andrew ;)) is my family, who make the most important stuff meaningful: my wife Rebecca Moroney, my daughter Claudia Moroney, and my son Christopher Moroney. Thanks to you all for making life more amazing than I ever thought it could be.

PART I

Building Models

Introduction to TensorFlow

When it comes to creating artificial intelligence (AI), machine learning (ML) and deep learning are a great place to begin. When getting started, however, it's easy to get overwhelmed by the options and all the new terminology. This book aims to demystify things for programmers, taking you through writing code to implement concepts of machine learning and deep learning; and building models that behave more as a human does, with scenarios like computer vision, natural language processing (NLP), and more. Thus, they become a form of synthesized, or artificial, intelligence.

But when we refer to *machine learning*, what in fact is this phenomenon? Let's take a quick look at that, and consider it from a programmer's perspective before we go any further. After that, this chapter will show you how to install the tools of the trade, from TensorFlow itself to environments where you can code and debug your TensorFlow models.

What Is Machine Learning?

Before we get into the ins and outs of ML, let's consider how it evolved from traditional programming. We'll start by examining what traditional programming is, then consider cases where it is limited. Then we'll see how ML evolved to handle those cases, and as a result has opened up new opportunities to implement new scenarios, unlocking many of the concepts of artificial intelligence.

Traditional programming involves us writing rules, expressed in a programming language, that act on data and give us answers. This applies just about everywhere that something can be programmed with code.

For example, consider a game like the popular Breakout. Code determines the movement of the ball, the score, and the various conditions for winning or losing the game. Think about the scenario where the ball bounces off a brick, like in [Figure 1-1](#).

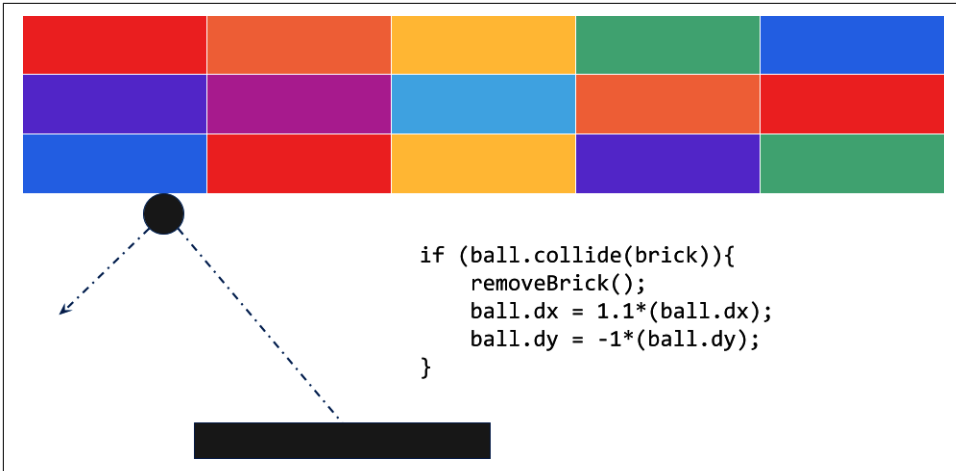


Figure 1-1. Code in a Breakout game

Here, the motion of the ball can be determined by its `dx` and `dy` properties. When it hits a brick, the brick is removed, and the velocity of the ball increases and changes direction. The code acts on data about the game situation.

Alternatively, consider a financial services scenario. You have data about a company's stock, such as its current price and current earnings. You can calculate a valuable ratio called the P/E (for price divided by earnings) with code like that in Figure 1-2.

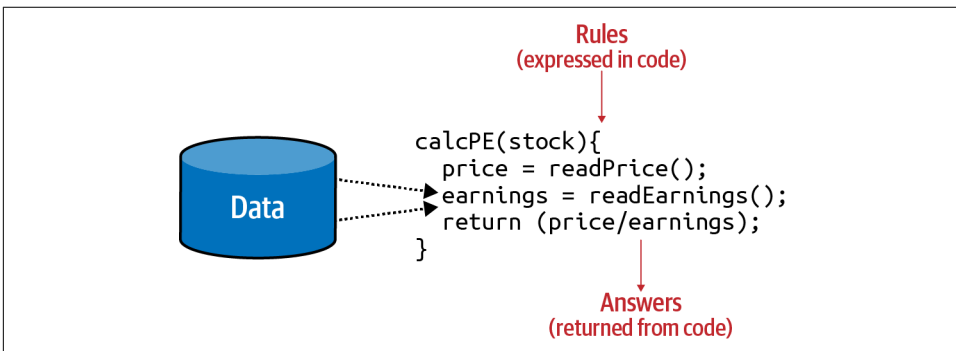


Figure 1-2. Code in a financial services scenario

Your code reads the price, reads the earnings, and returns a value that is the former divided by the latter.

If I were to try to sum up traditional programming like this into a single diagram, it might look like **Figure 1-3**.



Figure 1-3. High-level view of traditional programming

As you can see, you have rules expressed in a programming language. These rules act on data, and the result is answers.

Limitations of Traditional Programming

The model from **Figure 1-3** has been the backbone of development since its inception. But it has an inherent limitation: namely, that only scenarios that can be implemented are ones for which you can derive rules. What about other scenarios? Usually, they are infeasible to develop because the code is too complex. It's just not possible to write code to handle them.

Consider, for example, activity detection. Fitness monitors that can detect our activity are a recent innovation, not just because of the availability of cheap and small hardware, but also because the algorithms to handle detection weren't previously feasible. Let's explore why.

Figure 1-4 shows a naive activity detection algorithm for walking. It can consider the person's speed. If it's less than a particular value, we can determine that they are probably walking.

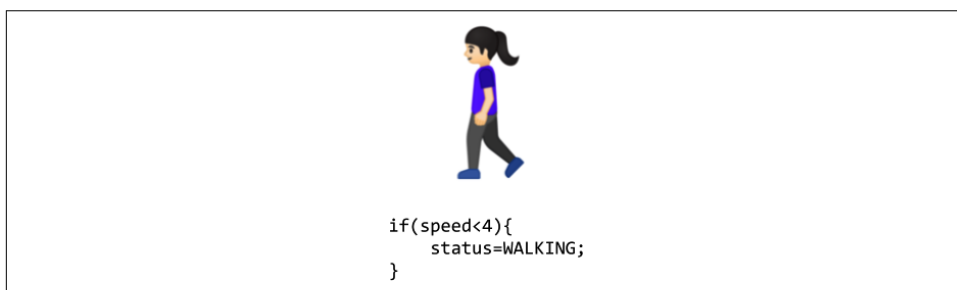


Figure 1-4. Algorithm for activity detection

Given that our data is speed, we could extend this to detect if they are running (Figure 1-5).

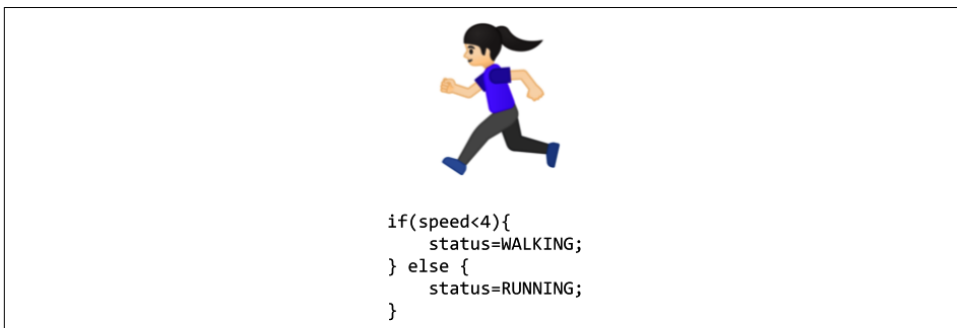


Figure 1-5. Extending the algorithm for running

As you can see, going by the speed, we might say if it is less than a particular value (say, 4 mph) the person is walking, and otherwise they are running. It still sort of works.

Now suppose we want to extend this to another popular fitness activity, biking. The algorithm could look like Figure 1-6.

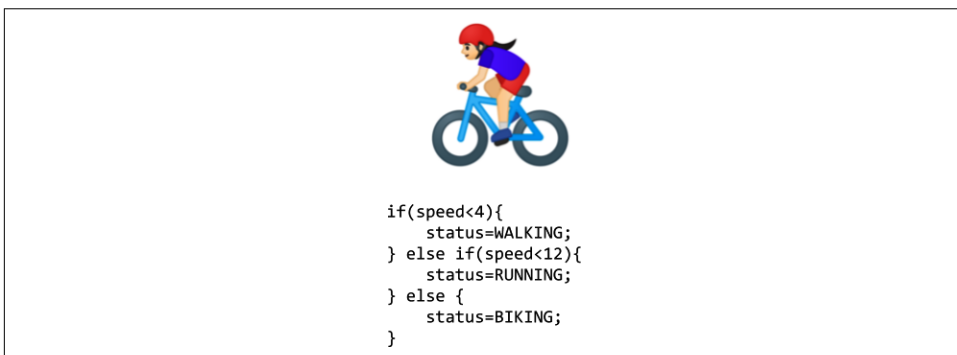


Figure 1-6. Extending the algorithm for biking

I know it's naive in that it just detects speed—some people run faster than others, and you might run downhill faster than you cycle uphill, for example. But on the whole, it still works. However, what happens if we want to implement another scenario, such as golfing (Figure 1-7)?

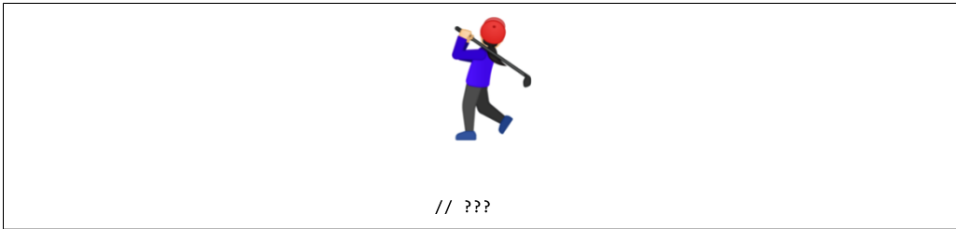


Figure 1-7. How do we write a golfing algorithm?

We're now stuck. How do we determine that someone is golfing using this methodology? The person might walk for a bit, stop, do some activity, walk for a bit more, stop, etc. But how can we tell this is golf?

Our ability to detect this activity using traditional rules has hit a wall. But maybe there's a better way.

Enter machine learning.

From Programming to Learning

Let's look back at the diagram that we used to demonstrate what traditional programming is (Figure 1-8). Here we have rules that act on data and give us answers. In our activity detection scenario, the data was the speed at which the person was moving; from that we could write rules to detect their activity, be it walking, biking, or running. We hit a wall when it came to golfing, because we couldn't come up with rules to determine what that activity looks like.



Figure 1-8. The traditional programming flow

But what would happen if we were to flip the axes around on this diagram? Instead of us coming up with the *rules*, what if we were to come up with the *answers*, and along with the data have a way of figuring out what the rules might be?

Figure 1-9 shows what this would look like. We can consider this high-level diagram to define *machine learning*.



Figure 1-9. Changing the axes to get machine learning

So what are the implications of this? Well, now instead of *us* trying to figure out what the rules are, we get lots of data about our scenario, we label that data, and the computer can figure out what the rules are that make one piece of data match a particular label and another piece of data match a different label.

How would this work for our activity detection scenario? Well, we can look at all the sensors that give us data about this person. If they have a wearable that detects information such as heart rate, location, speed, etc.—and if we collect a lot of instances of this data while they’re doing different activities—we end up with a scenario of having data that says “This is what walking looks like,” “This is what running looks like,” and so on (Figure 1-10).

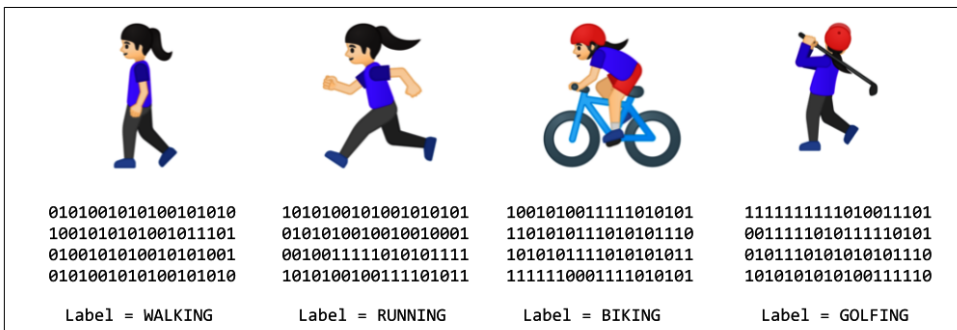


Figure 1-10. From coding to ML: gathering and labeling data

Now our job as programmers changes from figuring out the rules, to determining the activities, to writing the code that matches the data to the labels. If we can do this, then we can expand the scenarios that we can implement with code. Machine learning is a technique that enables us to do this, but in order to get started, we’ll need a framework—and that’s where TensorFlow enters the picture. In the next section we’ll take a look at what it is and how to install it, and then later in this chapter you’ll write your first code that learns the pattern between two values, like in the preceding scenario. It’s a simple “Hello World” scenario, but it has the same foundational code pattern that’s used in extremely complex ones.

The field of artificial intelligence is large and abstract, encompassing everything to do with making computers think and act the way human beings do. One of the ways a human takes on new behaviors is through learning by example. The discipline of

machine learning can thus be thought of as an on-ramp to the development of artificial intelligence. Through it, a machine can learn to see like a human (a field called *computer vision*), read text like a human (natural language processing), and much more. We'll be covering the basics of machine learning in this book, using the TensorFlow framework.

What Is TensorFlow?

TensorFlow is an open source platform for creating and using machine learning models. It implements many of the common algorithms and patterns needed for machine learning, saving you from needing to learn all the underlying math and logic and enabling you to just to focus on your scenario. It's aimed at everyone from hobbyists, to professional developers, to researchers pushing the boundaries of artificial intelligence. Importantly, it also supports deployment of models to the web, cloud, mobile, and embedded systems. We'll be covering each of these scenarios in this book.

The high-level architecture of TensorFlow can be seen in [Figure 1-11](#).

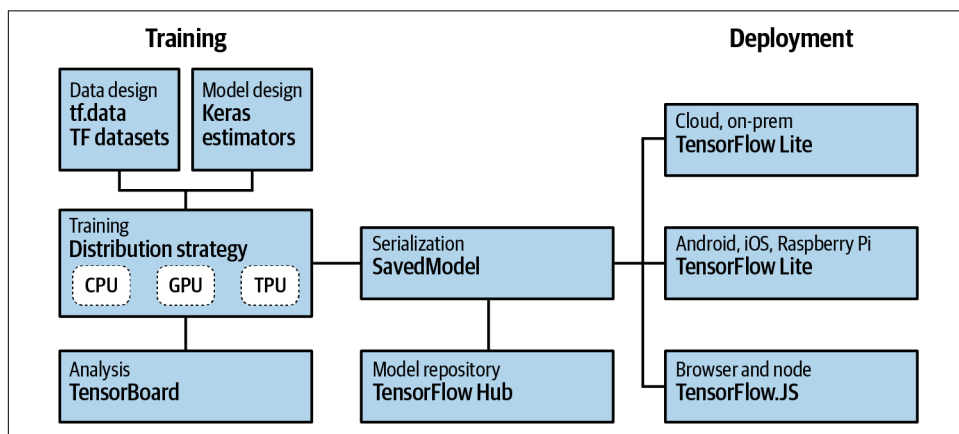


Figure 1-11. TensorFlow high-level architecture

The process of creating machine learning models is called *training*. This is where a computer uses a set of algorithms to learn about inputs and what distinguishes them from each other. So, for example, if you want a computer to recognize cats and dogs, you can use lots of pictures of both to create a model, and the computer will use that model to try to figure out what makes a cat a cat, and what makes a dog a dog. Once the model is trained, the process of having it recognize or categorize future inputs is called *inference*.

So, for training models, there are several things that you need to support. First is a set of APIs for designing the models themselves. With TensorFlow there are three main ways to do this: you can code everything by hand, where you figure out the logic for

how a computer can learn and then implement that in code (not recommended); you can use built-in *estimators*, which are already-implemented neural networks that you can customize; or you can use Keras, a high-level API that allows you to encapsulate common machine learning paradigms in code. This book will primarily focus on using the Keras APIs when creating models.

There are many ways to train a model. For the most part, you'll probably just use a single chip, be it a central processing unit (CPU), a graphics processing unit (GPU), or something new called a *tensor processing unit* (TPU). In more advanced working and research environments, parallel training across multiple chips can be used, employing a *distribution strategy* where training spans multiple chips. TensorFlow supports this too.

The lifeblood of any model is its data. As discussed earlier, if you want to create a model that can recognize cats and dogs, it needs to be trained with lots of examples of cats and dogs. But how can you manage these examples? You'll see, over time, that this can often involve a lot more coding than the creation of the models themselves. TensorFlow ships with APIs to try to ease this process, called TensorFlow Data Services. For learning, they include lots of preprocessed datasets that you can use with a single line of code. They also give you the tools for processing raw data to make it easier to use.

Beyond creating models, you'll also need to be able to get them into people's hands where they can be used. To this end, TensorFlow includes APIs for *serving*, where you can provide model inference over an HTTP connection for cloud or web users. For models to run on mobile or embedded systems, there's TensorFlow Lite, which provides tools for model inference on Android and iOS as well as Linux-based embedded systems such as a Raspberry Pi. A fork of TensorFlow Lite, called TensorFlow Lite Micro (TFLM), also provides inference on microcontrollers through an emerging concept known as TinyML. Finally, if you want to provide models to your browser or Node.js users, TensorFlow.js offers the ability to train and execute models in this way.

Next, I'll show you how to install TensorFlow so that you can get started creating and using ML models with it.

Using TensorFlow

In this section, we'll look at the three main ways that you can install and use TensorFlow. We'll start with how to install it on your developer box using the command line. We'll then explore using the popular PyCharm IDE (integrated development environment) to install TensorFlow. Finally, we'll look at Google Colab and how it can be used to access TensorFlow code with a cloud-based backend in your browser.

Installing TensorFlow in Python

TensorFlow supports the creation of models using multiple languages, including Python, Swift, Java, and more. In this book we'll focus on using Python, which is the de facto language for machine learning due to its extensive support for mathematical models. If you don't have it already, I strongly recommend you visit [Python](#) to get up and running with it, and [learnpython.org](#) to learn the Python language syntax.

With Python there are many ways to install frameworks, but the default one supported by the TensorFlow team is pip.

So, in your Python environment, installing TensorFlow is as easy as using:

```
pip install tensorflow
```

Note that starting with version 2.1, this will install the GPU version of TensorFlow by default. Prior to that, it used the CPU version. So, before installing, make sure you have a supported GPU and all the requisite drivers for it. Details on this are available at [TensorFlow](#).

If you don't have the required GPU or drivers, you can still install the CPU version of TensorFlow on any Linux, PC, or Mac with:

```
pip install tensorflow-cpu
```

Once you're up and running, you can test your TensorFlow version with the following code:

```
import tensorflow as tf
print(tf.__version__)
```

You should see output like that in [Figure 1-12](#). This will print the currently running version of TensorFlow—here you can see that version 2.0.0 is installed.

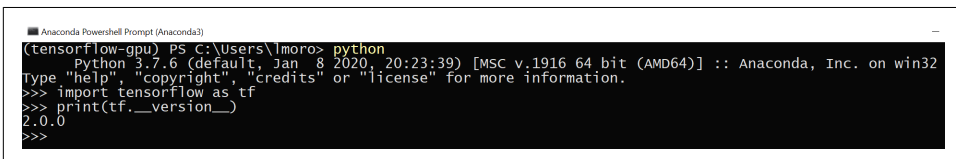
A screenshot of an Anaconda PowerShell Prompt window. The title bar reads "Anaconda PowerShell Prompt (Anaconda3)". The command prompt shows the user running 'python' in a directory 'PS C:\Users\lmoro>'. The output shows 'Python 3.7.6 (default, Jan 8 2020, 20:23:39) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32' followed by a help message. Then, the user enters '>>> import tensorflow as tf' and '>>> print(tf.__version__)', which outputs '2.0.0'.

Figure 1-12. Running TensorFlow in Python

Using TensorFlow in PyCharm

I'm particularly fond of using the [free community version of PyCharm](#) for building models using TensorFlow. PyCharm is useful for many reasons, but one of my favorites is that it makes the management of virtual environments easy. This means you can have Python environments with versions of tools such as TensorFlow that are specific to your particular project. So, for example, if you want to use TensorFlow 2.0 in one project and TensorFlow 2.1 in another, you can separate these with virtual

environments and not have to deal with installing/uninstalling dependencies when you switch. Additionally, with PyCharm you can do step-by-step debugging of your Python code—a must, especially if you’re just getting started.

For example, in [Figure 1-13](#) I have a new project that is called *example1*, and I’m specifying that I am going to create a new environment using Conda. When I create the project I’ll have a clean new virtual Python environment into which I can install any version of TensorFlow I want.

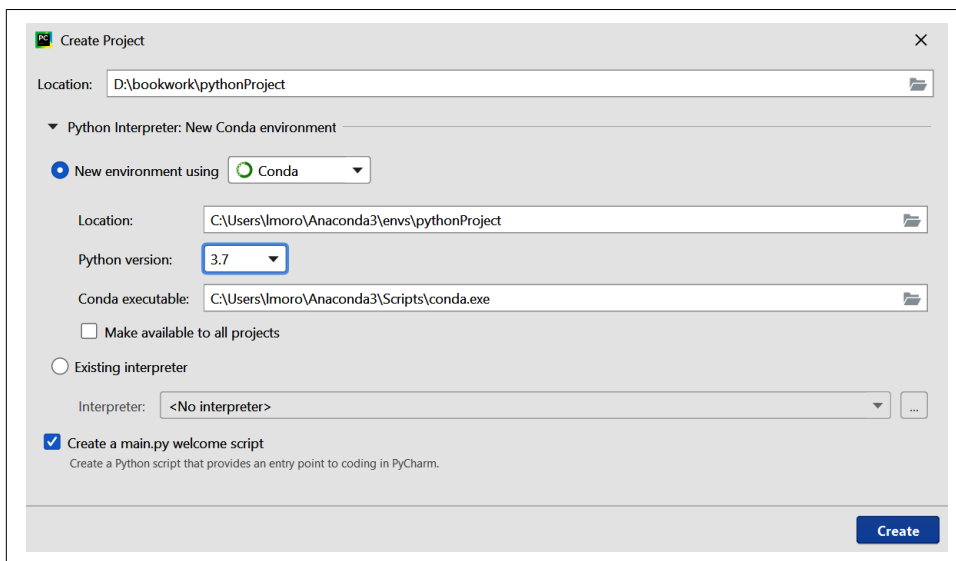


Figure 1-13. Creating a new virtual environment using PyCharm

Once you’ve created a project, you can open the File → Settings dialog and choose the entry for “Project: *<your project name>*” from the menu on the left. You’ll then see choices to change the settings for the Project Interpreter and the Project Structure. Choose the Project Interpreter link, and you’ll see the interpreter that you’re using, as well as a list of packages that are installed in this virtual environment ([Figure 1-14](#)).

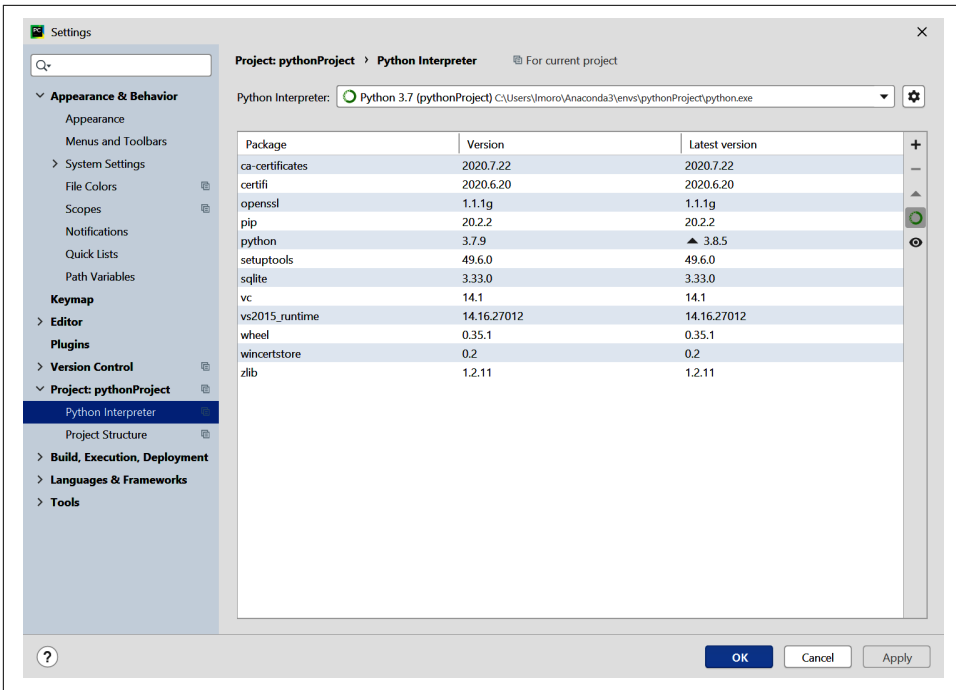


Figure 1-14. Adding packages to a virtual environment

Click the + button on the right, and a dialog will open showing the packages that are currently available. Type “tensorflow” into the search box and you’ll see all available packages with “tensorflow” in the name (Figure 1-15).

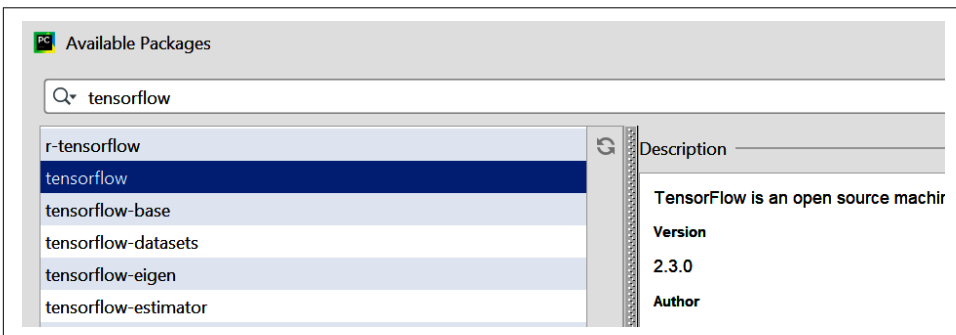


Figure 1-15. Installing TensorFlow with PyCharm

Once you’ve selected TensorFlow, or any other package you want to install, and then click the Install Package button, PyCharm will do the rest.

Once TensorFlow is installed, you can now write and debug your TensorFlow code in Python.

Using TensorFlow in Google Colab

Another option, which is perhaps easiest for getting started, is to use [Google Colab](#), a hosted Python environment that you can access via a browser. What's really neat about Colab is that it provides GPU and TPU backends so you can train models using state-of-the-art hardware at no cost.

When you visit the Colab website, you'll be given the option to open previous Colabs or start a new notebook, as shown in [Figure 1-16](#).

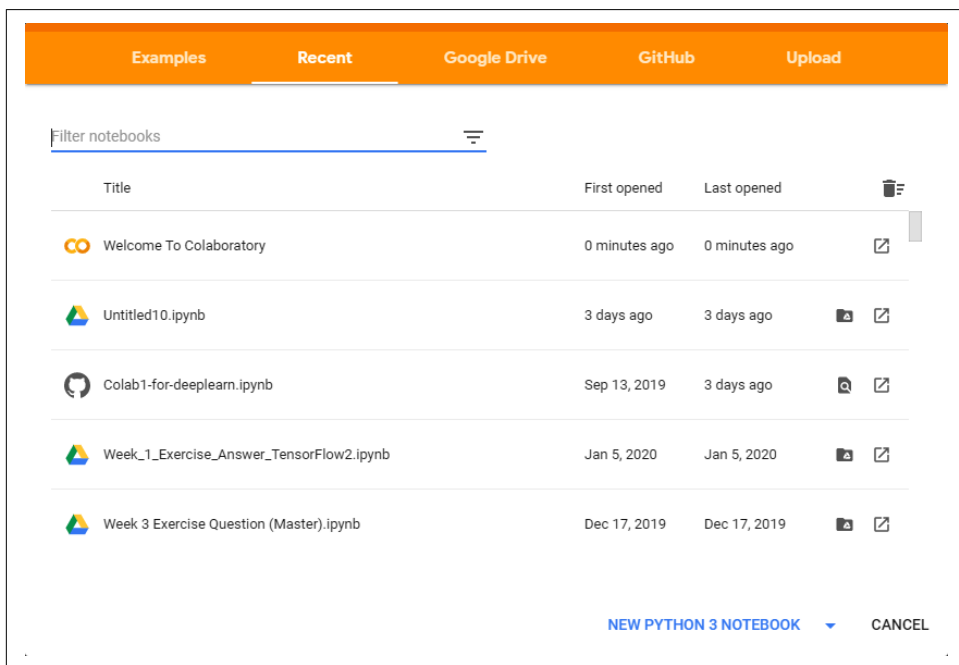


Figure 1-16. Getting started with Google Colab

Clicking the New Python 3 Notebook link will open the editor, where you can add panes of code or text ([Figure 1-17](#)). You can execute the code by clicking the Play button (the arrow) to the left of the pane.

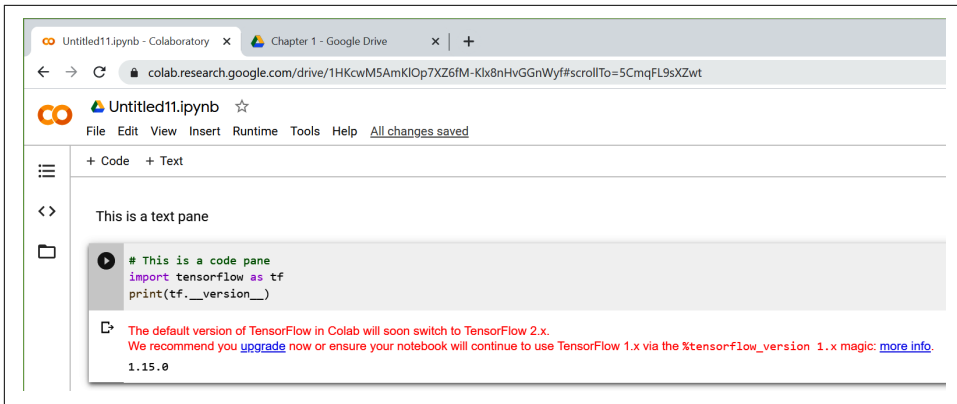


Figure 1-17. Running TensorFlow code in Colab

It's always a good idea to check the TensorFlow version, as shown here, to be sure you're running the correct version. Often Colab's built-in TensorFlow will be a version or two behind the latest release. If that's the case you can update it with `pip install` as shown earlier, by simply using a block of code like this:

```
!pip install tensorflow==2.1.0
```

Once you run this command, your current environment within Colab will use the desired version of TensorFlow.

Getting Started with Machine Learning

As we saw earlier in the chapter, the machine learning paradigm is one where you have data, that data is labeled, and you want to figure out the rules that match the data to the labels. The simplest possible scenario to show this in code is as follows. Consider these two sets of numbers:

```
X = -1, 0, 1, 2, 3, 4
Y = -3, -1, 1, 3, 5, 7
```

There's a relationship between the X and Y values (for example, if X is -1 then Y is -3, if X is 3 then Y is 5, and so on). Can you see it?

After a few seconds you probably saw that the pattern here is $Y = 2X - 1$. How did you get that? Different people work it out in different ways, but I typically hear the observation that X increases by 1 in its sequence, and Y increases by 2; thus, $Y = 2X$ +/- something. They then look at when $X = 0$ and see that $Y = -1$, so they figure that the answer could be $Y = 2X - 1$. Next they look at the other values and see that this hypothesis "fits," and the answer is $Y = 2X - 1$.

That's very similar to the machine learning process. Let's take a look at some TensorFlow code that you could write to have a neural network do this figuring out for you.

Here's the full code, using the TensorFlow Keras APIs. Don't worry if it doesn't make sense yet; we'll go through it line by line:

```
import tensorflow as tf
import numpy as np
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([Dense(units=1, input_shape=[1])])
model.compile(optimizer='sgd', loss='mean_squared_error')

xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

model.fit(xs, ys, epochs=500)

print(model.predict([10.0]))
```

Let's start with the first line. You've probably heard of neural networks, and you've probably seen diagrams that explain them using layers of interconnected neurons, a little like [Figure 1-18](#).

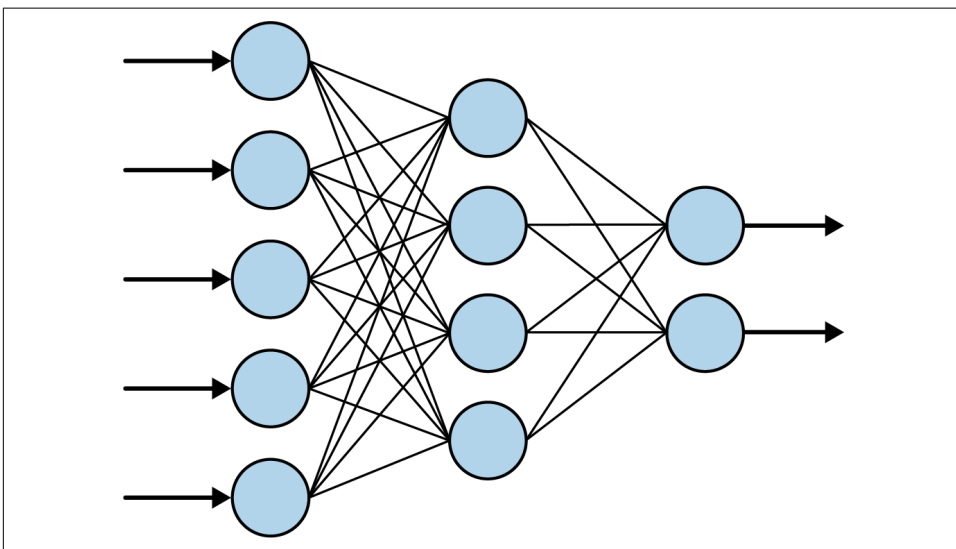


Figure 1-18. A typical neural network

When you see a neural network like this, consider each of the “circles” to be a *neuron*, and each of the columns of circles to be a *layer*. So, in [Figure 1-18](#), there are three layers: the first has five neurons, the second has four, and the third has two.

If we look back at our code and look at just the first line, we'll see that we're defining the simplest possible neural network. There's only one layer, and it contains only one neuron:

```
model = Sequential([Dense(units=1, input_shape=[1])])
```

When using TensorFlow, you define your layers using `Sequential`. Inside the `Sequential`, you then specify what each layer looks like. We only have one line inside our `Sequential`, so we have only one layer.

You then define what the layer looks like using the `keras.layers` API. There are lots of different layer types, but here we're using a `Dense` layer. "Dense" means a set of fully (or densely) connected neurons, which is what you can see in [Figure 1-18](#) where every neuron is connected to every neuron in the next layer. It's the most common form of layer type. Our `Dense` layer has `units=1` specified, so we have just one dense layer with one neuron in our entire neural network. Finally, when you specify the *first* layer in a neural network (in this case, it's our only layer), you have to tell it what the shape of the input data is. In this case our input data is our `X`, which is just a single value, so we specify that that's its shape.

The next line is where the fun really begins. Let's look at it again:

```
model.compile(optimizer='sgd', loss='mean_squared_error')
```

If you've done anything with machine learning before, you've probably seen that it involves a lot of mathematics. If you haven't done calculus in years it might have seemed like a barrier to entry. Here's the part where the math comes in—it's the core to machine learning.

In a scenario such as this one, the computer has *no idea* what the relationship between `X` and `Y` is. So it will make a guess. Say for example it guesses that $Y = 10X + 10$. It then needs to measure how good or how bad that guess is. That's the job of the *loss function*.

It already knows the answers when `X` is `-1`, `0`, `1`, `2`, `3`, and `4`, so the loss function can compare these to the answers for the guessed relationship. If it guessed $Y = 10X + 10$, then when `X` is `-1`, `Y` will be `0`. The correct answer there was `-3`, so it's a bit off. But when `X` is `4`, the guessed answer is `50`, whereas the correct one is `7`. That's really far off.

Armed with this knowledge, the computer can then make another guess. That's the job of the *optimizer*. This is where the heavy calculus is used, but with TensorFlow, that can be hidden from you. You just pick the appropriate optimizer to use for different scenarios. In this case we picked one called `sgd`, which stands for *stochastic gradient descent*—a complex mathematical function that, when given the values, the previous guess, and the results of calculating the errors (or loss) on that guess, can

then generate another one. Over time, its job is to minimize the loss, and by so doing bring the guessed formula closer and closer to the correct answer.

Next, we simply format our numbers into the data format that the layers expect. In Python, there's a library called Numpy that TensorFlow can use, and here we put our numbers into a Numpy array to make it easy to process them:

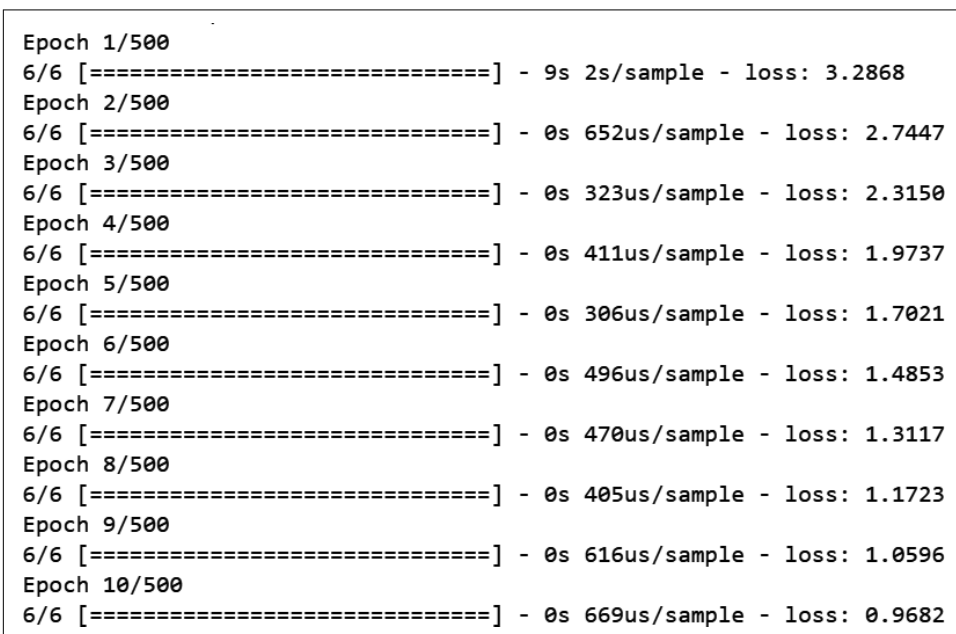
```
xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)
```

The *learning* process will then begin with the `model.fit` command, like this:

```
model.fit(xs, ys, epochs=500)
```

You can read this as “fit the Xs to the Ys, and try it 500 times.” So, on the first try, the computer will guess the relationship (i.e., something like $Y = 10X + 10$), and measure how good or bad that guess was. It will then feed those results to the optimizer, which will generate another guess. This process will then be repeated, with the logic being that the loss (or error) will go down over time, and as a result the “guess” will get better and better.

Figure 1-19 shows a screenshot of this running in a Colab notebook. Take a look at the loss values over time.



```
Epoch 1/500
6/6 [=====] - 9s 2s/sample - loss: 3.2868
Epoch 2/500
6/6 [=====] - 0s 652us/sample - loss: 2.7447
Epoch 3/500
6/6 [=====] - 0s 323us/sample - loss: 2.3150
Epoch 4/500
6/6 [=====] - 0s 411us/sample - loss: 1.9737
Epoch 5/500
6/6 [=====] - 0s 306us/sample - loss: 1.7021
Epoch 6/500
6/6 [=====] - 0s 496us/sample - loss: 1.4853
Epoch 7/500
6/6 [=====] - 0s 470us/sample - loss: 1.3117
Epoch 8/500
6/6 [=====] - 0s 405us/sample - loss: 1.1723
Epoch 9/500
6/6 [=====] - 0s 616us/sample - loss: 1.0596
Epoch 10/500
6/6 [=====] - 0s 669us/sample - loss: 0.9682
```

Figure 1-19. Training the neural network

We can see that over the first 10 epochs, the loss went from 3.2868 to 0.9682. That is, after only 10 tries, the network was performing three times better than with its initial guess. Then take a look at what happens by the five hundredth epoch (Figure 1-20).

```
Epoch 495/500
6/6 [=====] - 0s 374us/sample - loss: 2.9063e-05
Epoch 496/500
6/6 [=====] - 0s 540us/sample - loss: 2.8466e-05
Epoch 497/500
6/6 [=====] - 0s 382us/sample - loss: 2.7882e-05
Epoch 498/500
6/6 [=====] - 0s 397us/sample - loss: 2.7309e-05
Epoch 499/500
6/6 [=====] - 0s 367us/sample - loss: 2.6748e-05
Epoch 500/500
6/6 [=====] - 0s 363us/sample - loss: 2.6199e-05
```

Figure 1-20. Training the neural network—the last five epochs

We can now see the loss is 2.61×10^{-5} . The loss has gotten so small that the model has pretty much figured out that the relationship between the numbers is $Y = 2X - 1$. The *machine* has *learned* the pattern between them.

Our last line of code then used the trained model to get a prediction like this:

```
print(model.predict([10.0]))
```



The term *prediction* is typically used when dealing with ML models. Don't think of it as looking into the future, though! This term is used because we're dealing with a certain amount of uncertainty. Think back to the activity detection scenario we spoke about earlier. When the person was moving at a certain speed, she was *probably* walking. Similarly, when a model learns about the patterns between two things it will tell us what the answer *probably* is. In other words, it is *predicting* the answer. (Later you'll also learn about *inference*, where the model is picking one answer among many, and *inferring* that it has picked the correct one.)

What do you think the answer will be when we ask the model to predict Y when X is 10? You might instantly think 19, but that's not correct. It will pick a value *very close* to 19. There are several reasons for this. First of all, our loss wasn't 0. It was still a very small amount, so we should expect any predicted answer to be off by a very small amount. Secondly, the neural network is trained on only a small amount of data—in this case only six pairs of (X,Y) values.

The model only has a single neuron in it, and that neuron learns a *weight* and a *bias*, so that $Y = WX + B$. This looks exactly like the relationship $Y = 2X - 1$ that we want,

where we would want it to learn that $W = 2$ and $B = -1$. Given that the model was trained on only six items of data, the answer could never be expected to be exactly these values, but something very close to them.

Run the code for yourself to see what you get. I got 18.977888 when I ran it, but your answer may differ slightly because when the neural network is first initialized there's a random element: your initial guess will be slightly different from mine, and from a third person's.

Seeing What the Network Learned

This is obviously a very simple scenario, where we are matching X s to Y s in a linear relationship. As mentioned in the previous section, neurons have weight and bias parameters that they learn, which makes a single neuron fine for learning a relationship like this: namely, when $Y = 2X - 1$, the weight is 2 and the bias is -1 . With TensorFlow we can actually take a look at the weights and biases that are learned, with a simple change to our code like this:

```
import tensorflow as tf
import numpy as np
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

l0 = Dense(units=1, input_shape=[1])
model = Sequential([l0])
model.compile(optimizer='sgd', loss='mean_squared_error')

xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

model.fit(xs, ys, epochs=500)

print(model.predict([10.0]))
print("Here is what I learned: {}".format(l0.get_weights()))
```

The difference is that I created a variable called `l0` to hold the Dense layer. Then, after the network finishes learning, I can print out the values (or weights) that the layer learned.

In my case, the output was as follows:

```
Here is what I learned: [array([[1.9967953]], dtype=float32),
array([-0.9900647], dtype=float32)]
```

Thus, the learned relationship between X and Y was $Y = 1.9967953X - 0.9900647$.

This is pretty close to what we'd expect ($Y = 2X - 1$), and we could argue that it's even closer to reality, because we are *assuming* that the relationship will hold for other values.

Summary

That's it for your first "Hello World" of machine learning. You might be thinking that this seems like massive overkill for something as simple as determining a linear relationship between two values. And you'd be right. But the cool thing about this is that the pattern of code we've created here is the same pattern that's used for far more complex scenarios. You'll see these starting in [Chapter 2](#), where we'll explore some basic computer vision techniques—the machine will learn to “see” patterns in pictures, and identify what's in them.

Introduction to Computer Vision

The previous chapter introduced the basics of how machine learning works. You saw how to get started with programming using neural networks to match data to labels, and from there how to infer the rules that can be used to distinguish items. A logical next step is to apply these concepts to computer vision, where we will have a model learn how to recognize content in pictures so it can “see” what’s in them. In this chapter you’ll work with a popular dataset of clothing items and build a model that can differentiate between them, thus “seeing” the difference between different types of clothing.

Recognizing Clothing Items

For our first example, let’s consider what it takes to recognize items of clothing in an image. Consider, for example, the items in [Figure 2-1](#).



Figure 2-1. Examples of clothing

There are a number of different clothing items here, and you can recognize them. You understand what is a shirt, or a coat, or a dress. But how would you explain this to somebody who has never seen clothing? How about a shoe? There are two shoes in

this image, but how would you describe that to somebody? This is another area where the rules-based programming we spoke about in [Chapter 1](#) can fall down. Sometimes it's just infeasible to describe something with rules.

Of course, computer vision is no exception. But consider how you learned to recognize all these items—by seeing lots of different examples, and gaining experience with how they're used. Can we do the same with a computer? The answer is yes, but with limitations. Let's take a look at a first example of how to teach a computer to recognize items of clothing, using a well-known dataset called Fashion MNIST.

The Data: Fashion MNIST

One of the foundational datasets for learning and benchmarking algorithms is the Modified National Institute of Standards and Technology (MNIST) database, by Yann LeCun, Corinna Cortes, and Christopher Burges. This dataset is comprised of images of 70,000 handwritten digits from 0 to 9. The images are 28×28 grayscale.

[Fashion MNIST](#) is designed to be a drop-in replacement for MNIST that has the same number of records, the same image dimensions, and the same number of classes—so, instead of images of the digits 0 through 9, Fashion MNIST contains images of 10 different types of clothing. You can see an example of the contents of the dataset in [Figure 2-2](#). Here, three lines are dedicated to each clothing item type.

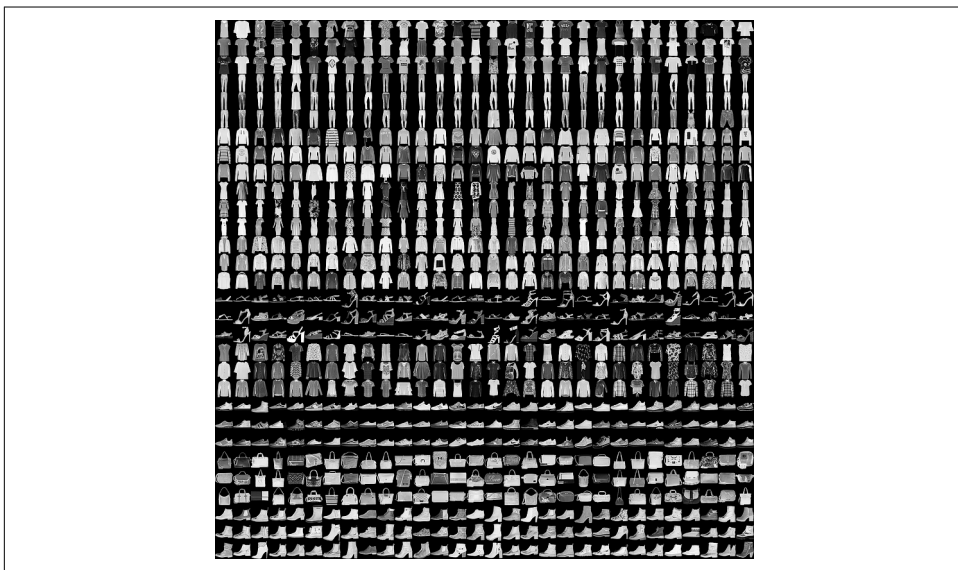


Figure 2-2. Exploring the Fashion MNIST dataset

It has a nice variety of clothing, including shirts, trousers, dresses, and lots of types of shoes. As you may notice, it's monochrome, so each picture consists of a certain number of pixels with values between 0 and 255. This makes the dataset simpler to manage.

You can see a closeup of a particular image from the dataset in [Figure 2-3](#).

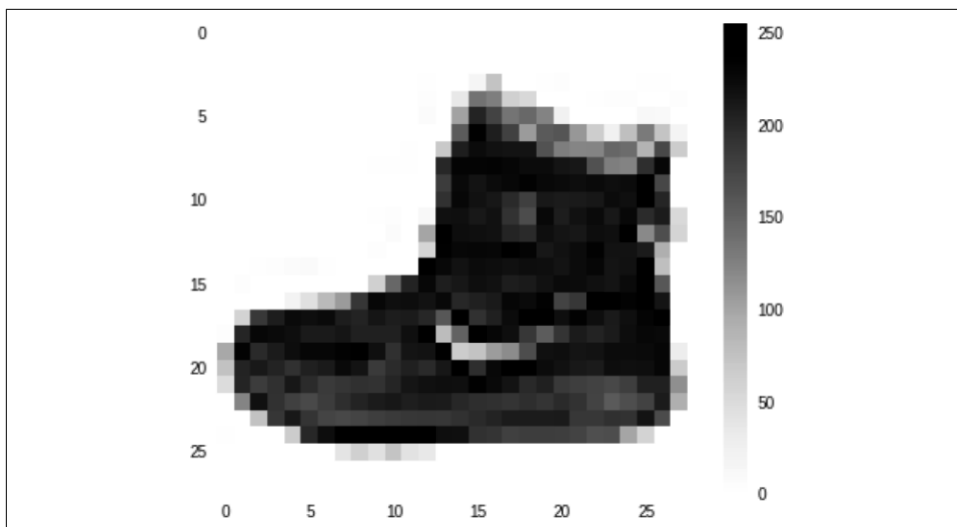


Figure 2-3. Closeup of an image in the Fashion MNIST dataset

Like any image, it's a rectangular grid of pixels. In this case the grid size is 28×28 , and each pixel is simply a value between 0 and 255, as mentioned previously. Let's now take a look at how you can use these pixel values with the functions we saw previously.

Neurons for Vision

In [Chapter 1](#), you saw a very simple scenario where a machine was given a set of X and Y values, and it learned that the relationship between these was $Y = 2X - 1$. This was done using a very simple neural network with one layer and one neuron.

If you were to draw that visually, it might look like [Figure 2-4](#).

Each of our images is a set of 784 values (28×28) between 0 and 255. They can be our X. We know that we have 10 different types of images in our dataset, so let's consider them to be our Y. Now we want to learn what the function looks like where Y is a function of X.

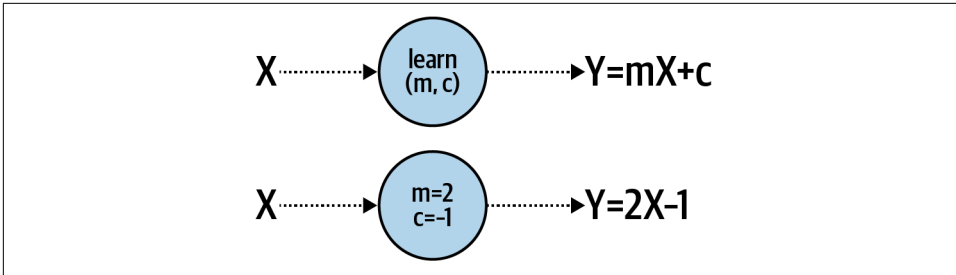


Figure 2-4. A single neuron learning a linear relationship

Given that we have 784 X values per image, and our Y is going to be between 0 and 9, it's pretty clear that we cannot do $Y = mX + c$ as we did earlier.

But what we *can* do is have several neurons working together. Each of these will learn *parameters*, and when we have a combined function of all of these parameters working together, we can see if we can match that pattern to our desired answer (Figure 2-5).

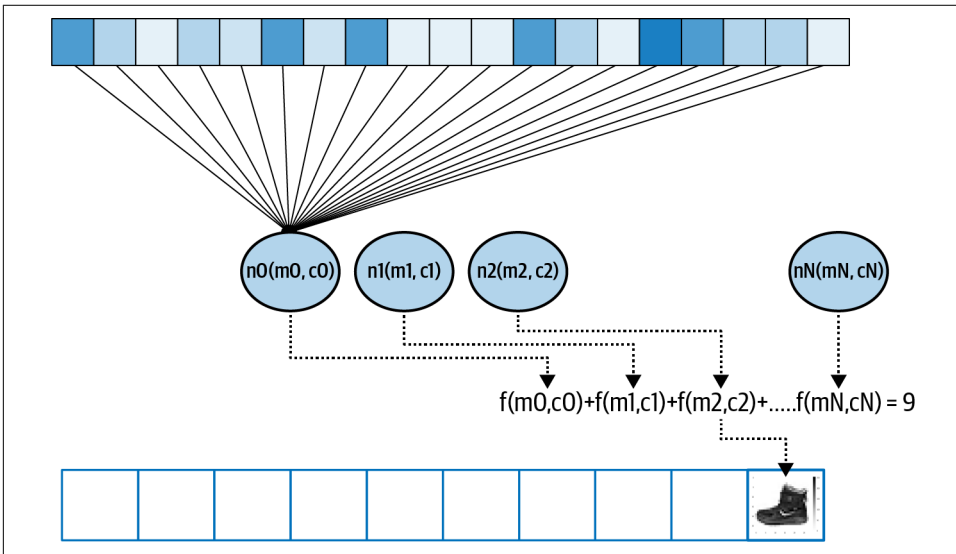


Figure 2-5. Extending our pattern for a more complex example

The boxes at the top of this diagram can be considered the pixels in the image, or our X values. When we train the neural network we load these into a layer of neurons—Figure 2-5 shows them just being loaded into the first neuron, but the values are loaded into each of them. Consider each neuron's weight and bias (m and c) to be randomly initialized. Then, when we sum up the values of the output of each neuron we're going to get a value. This will be done for *every* neuron in the output layer, so

neuron 0 will contain the value of the probability that the pixels add up to label 0, neuron 1 for label 1, etc.

Over time, we want to match that value to the desired output—which for this image we can see is the number 9, the label for the ankle boot shown in [Figure 2-3](#). So, in other words, this neuron should have the largest value of all of the output neurons.

Given that there are 10 labels, a random initialization should get the right answer about 10% of the time. From that, the loss function and optimizer can do their job epoch by epoch to tweak the internal parameters of each neuron to improve that 10%. And thus, over time, the computer will learn to “see” what makes a shoe a shoe or a dress a dress.

Designing the Neural Network

Let’s now explore what this looks like in code. First, we’ll look at the design of the neural network shown in [Figure 2-5](#):

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

If you remember, in [Chapter 1](#) we had a `Sequential` model to specify that we had many layers. It only had one layer, but in this case, we have multiple layers.

The first, `Flatten`, isn’t a layer of neurons, but an input layer specification. Our inputs are 28×28 images, but we want them to be treated as a series of numeric values, like the gray boxes at the top of [Figure 2-5](#). `Flatten` takes that “square” value (a 2D array) and turns it into a line (a 1D array).

The next one, `Dense`, is a layer of neurons, and we’re specifying that we want 128 of them. This is the middle layer shown in [Figure 2-5](#). You’ll often hear such layers described as *hidden layers*. Layers that are between the inputs and the outputs aren’t seen by a caller, so the term “hidden” is used to describe them. We’re asking for 128 neurons to have their internal parameters randomly initialized. Often the question I’ll get asked at this point is “Why 128?” This is entirely arbitrary—there’s no fixed rule for the number of neurons to use. As you design the layers you want to pick the appropriate number of values to enable your model to actually learn. More neurons means it will run more slowly, as it has to learn more parameters. More neurons could also lead to a network that is great at recognizing the training data, but not so good at recognizing data that it hasn’t previously seen (this is known as *overfitting*, and we’ll discuss it later in this chapter). On the other hand, fewer neurons means that the model might not have sufficient parameters to learn.

It takes some experimentation over time to pick the right values. This process is typically called *hyperparameter tuning*. In machine learning, a hyperparameter is a value that is used to control the training, as opposed to the internal values of the neurons that get trained/learned, which are referred to as parameters.

You might notice that there's also an *activation function* specified in that layer. The activation function is code that will execute on each neuron in the layer. TensorFlow supports a number of them, but a very common one in middle layers is `relu`, which stands for *rectified linear unit*. It's a simple function that just returns a value if it's greater than 0. In this case, we don't want negative values being passed to the next layer to potentially impact the summing function, so instead of writing a lot of `if-then` code, we can simply activate the layer with `relu`.

Finally, there's another Dense layer, which is the output layer. This has 10 neurons, because we have 10 classes. Each of these neurons will end up with a probability that the input pixels match that class, so our job is to determine which one has the highest value. We could loop through them to pick that value, but the `softmax` activation function does that for us.

So now when we train our neural network, the goal is that we can feed in a 28×28 -pixel array and the neurons in the middle layer will have weights and biases (m and c values) that when combined will match those pixels to one of the 10 output values.

The Complete Code

Now that we've explored the architecture of the neural network, let's look at the complete code for training one with the Fashion MNIST data:

```
import tensorflow as tf
data = tf.keras.datasets.fashion_mnist

(training_images, training_labels), (test_images, test_labels) = data.load_data()

training_images = training_images / 255.0
test_images = test_images / 255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(training_images, training_labels, epochs=5)
```

Let's walk through this piece by piece. First is a handy shortcut for accessing the data:

```
data = tf.keras.datasets.fashion_mnist
```

Keras has a number of built-in datasets that you can access with a single line of code like this. In this case you don't have to handle downloading the 70,000 images—splitting them into training and test sets, and so on—all it takes is one line of code. This methodology has been improved upon using an API called **TensorFlow Datasets**, but for the purposes of these early chapters, to reduce the number of new concepts you need to learn, we'll just use `tf.keras.datasets`.

We can call its `load_data` method to return our training and test sets like this:

```
(training_images, training_labels),  
(test_images, test_labels) = data.load_data()
```

Fashion MNIST is designed to have 60,000 training images and 10,000 test images. So, the return from `data.load_data` will give you an array of 60,000 28×28 -pixel arrays called `training_images`, and an array of 60,000 values (0–9) called `training_labels`. Similarly, the `test_images` array will contain 10,000 28×28 -pixel arrays, and the `test_labels` array will contain 10,000 values between 0 and 9.

Our job will be to fit the training images to the training labels in a similar manner to how we fit Y to X in **Chapter 1**.

We'll hold back the test images and test labels so that the network does not see them while training. These can be used to test the efficacy of the network with hitherto unseen data.

The next lines of code might look a little unusual:

```
training_images = training_images / 255.0  
test_images = test_images / 255.0
```

Python allows you to do an operation across the entire array with this notation. Recall that all of the pixels in our images are grayscale, with values between 0 and 255. Dividing by 255 thus ensures that every pixel is represented by a number between 0 and 1 instead. This process is called *normalizing* the image.

The math for why normalized data is better for training neural networks is beyond the scope of this book, but bear in mind when training a neural network in TensorFlow that normalization will improve performance. Often your network will not learn and will have massive errors when dealing with non-normalized data. The $Y = 2X - 1$ example from **Chapter 1** didn't require the data to be normalized because it was very simple, but for fun try training it with different values of X and Y where X is much larger and you'll see it quickly fail!

Next we define the neural network that makes up our model, as discussed earlier:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

When we compile our model we specify the loss function and the optimizer as before:

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

The loss function in this case is called *sparse categorical cross entropy*, and it's one of the arsenal of loss functions that are built into TensorFlow. Again, choosing which loss function to use is an art in itself, and over time you'll learn which ones are best to use in which scenarios. One major difference between this model and the one we created in [Chapter 1](#) is that instead of us trying to predict a single number, here we're picking a *category*. Our item of clothing will belong to 1 of 10 categories of clothing, and thus using a *categorical* loss function is the way to go. Sparse categorical cross entropy is a good choice.

The same applies to choosing an optimizer. The *adam* optimizer is an evolution of the stochastic gradient descent (*sgd*) optimizer we used in [Chapter 1](#) that has been shown to be faster and more efficient. As we're handling 60,000 training images, any performance improvement we can get will be helpful, so that one is chosen here.

You might notice that a new line specifying the metrics we want to report is also present in this code. Here, we want to report back on the accuracy of the network as we're training. The simple example in [Chapter 1](#) just reported on the loss, and we interpreted that the network was learning by looking at how the loss was reduced. In this case, it's more useful to us to see how the network is learning by looking at the accuracy—where it will return how often it correctly matched the input pixels to the output label.

Next, we'll train the network by fitting the training images to the training labels over five epochs:

```
model.fit(training_images, training_labels, epochs=5)
```

Finally, we can do something new—evaluate the model, using a single line of code. We have a set of 10,000 images and labels for testing, and we can pass them to the trained model to have it predict what it thinks each image is, compare that to its actual label, and sum up the results:

```
model.evaluate(test_images, test_labels)
```

Training the Neural Network

Execute the code, and you'll see the network train epoch by epoch. After running the training, you'll see something at the end that looks like this:

```
58016/60000 [====>.] - ETA: 0s - loss: 0.2941 - accuracy: 0.8907
59552/60000 [====>.] - ETA: 0s - loss: 0.2943 - accuracy: 0.8906
60000/60000 [ ] - 2s 34us/sample - loss: 0.2940 - accuracy: 0.8906
```

Note that it's now reporting accuracy. So in this case, using the training data, our model ended up with an accuracy of about 89% after only five epochs.

But what about the test data? The results of `model.evaluate` on our test data will look something like this:

```
10000/1 [====] - 0s 30us/sample - loss: 0.2521 - accuracy: 0.8736
```

In this case the accuracy of the model was 87.36%, which isn't bad considering we only trained it for five epochs.

You're probably wondering why the accuracy is *lower* for the test data than it is for the training data. This is very commonly seen, and when you think about it, it makes sense: the neural network only really knows how to match the inputs it has been trained on with the outputs for those values. Our hope is that, given enough data, it will be able to generalize from the examples it has seen, "learning" what a shoe or a dress looks like. But there will always be examples of items that it hasn't seen that are sufficiently different from what it has to confuse it.

For example, if you grew up only ever seeing sneakers, and that's what a shoe looks like to you, when you first see a high heel you might be a little confused. From your experience, it's probably a shoe, but you don't know for sure. This is a similar concept.

Exploring the Model Output

Now that the model has been trained, and we have a good gage of its accuracy using the test set, let's explore it a little:

```
classifications = model.predict(test_images)
print(classifications[0])
print(test_labels[0])
```

We'll get a set of classifications by passing the test images to `model.predict`. Then let's see what we get if we print out the first of the classifications and compare it to the test label:

```
[1.9177722e-05 1.9856788e-07 6.3756357e-07 7.1702580e-08 5.5287035e-07
 1.2249852e-02 6.0708484e-05 7.3229447e-02 8.3050705e-05 9.1435629e-01]
9
```

You'll notice that the classification gives us back an array of values. These are the values of the 10 output neurons. The label is the actual label for the item of clothing, in this case 9. Take a look through the array—you'll see that some of the values are very small, and the last one (array index 9) is the largest by far. These are the probabilities that the image matches the label at that particular index. So, what the neural network is reporting is that there's a 91.4% chance that the item of clothing at index 0 is label 9. We know that it's label 9, so it got it right.

Try a few different values for yourself, and see if you can find anywhere the model gets it wrong.

Training for Longer—Discovering Overfitting

In this case, we trained for only five epochs. That is, we went through the entire training loop of having the neurons randomly initialized, checked against their labels, having that performance measured by the loss function, and then updated by the optimizer five times. And the results we got were pretty good: 89% accuracy on the training set and 87% on the test set. So what happens if we train for longer?

Try updating it to train for 50 epochs instead of 5. In my case, I got these accuracy figures on the training set:

```
58112/60000 [==>.] - ETA: 0s - loss: 0.0983 - accuracy: 0.9627
59520/60000 [==>.] - ETA: 0s - loss: 0.0987 - accuracy: 0.9627
60000/60000 [====] - 2s 35us/sample - loss: 0.0986 - accuracy: 0.9627
```

This is particularly exciting because we're doing much better: 96.27% accuracy. For the test set we reach 88.6%:

```
[====] - 0s 30us/sample - loss: 0.3870 - accuracy: 0.8860
```

So, we got a big improvement on the training set, and a smaller one on the test set. This might suggest that training our network for much longer would lead to much better results—but that's not always the case. The network is doing much better with the training data, but it's not necessarily a better model. In fact, the divergence in the accuracy numbers shows that it has become overspecialized to the training data, a process often called *overfitting*. As you build more neural networks this is something to watch out for, and as you go through this book you'll learn a number of techniques to avoid it.

Stopping Training

In each of the cases so far, we've hardcoded the number of epochs we're training for. While that works, we might want to train until we reach the desired accuracy instead of constantly trying different numbers of epochs and training and retraining until we get to our desired value. So, for example, if we want to train until the model is at 95%

accuracy on the training set, without knowing how many epochs that will take, how could we do that?

The easiest approach is to use a *callback* on the training. Let's take a look at the updated code that uses callbacks:

```
import tensorflow as tf

class myCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):
        if(logs.get('accuracy')>0.95):
            print("\nReached 95% accuracy so cancelling training!")
            self.model.stop_training = True

callbacks = myCallback()
mnist = tf.keras.datasets.fashion_mnist

(training_images, training_labels),
(test_images, test_labels) = mnist.load_data()

training_images=training_images/255.0
test_images=test_images/255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(training_images, training_labels, epochs=50,
          callbacks=[callbacks])
```

Let's see what we've changed here. First, we created a new class called `myCallback`. This takes a `tf.keras.callbacks.Callback` as a parameter. In it, we define the `on_epoch_end` function, which will give us details about the logs for this epoch. In these logs is an accuracy value, so all we have to do is see if it is greater than .95 (or 95%); if it is, we can stop training by saying `self.model.stop_training = True`.

Once we've specified this, we create a `callbacks` object to be an instance of the `myCallback` function.

Now check out the `model.fit` statement. You'll see that I've updated it to train for 50 epochs, and then added a `callbacks` parameter. To this, I pass the `callbacks` object.

When training, at the end of every epoch, the callback function will be called. So at the end of each epoch you'll check, and after about 34 epochs you'll see that your training will end, because the training has hit 95% accuracy (your number may be

slightly different because of the initial random initialization, but it will likely be quite close to 34):

```
56896/60000 [====>..] - ETA: 0s - loss: 0.1309 - accuracy: 0.9500
58144/60000 [====>..] - ETA: 0s - loss: 0.1308 - accuracy: 0.9502
59424/60000 [====>..] - ETA: 0s - loss: 0.1308 - accuracy: 0.9502
Reached 95% accuracy so cancelling training!
```

Summary

In [Chapter 1](#) you learned about how machine learning is based on fitting features to labels through sophisticated pattern matching with a neural network. In this chapter you took that to the next level, going beyond a single neuron, and learned how to create your first (very basic) computer vision neural network. It was somewhat limited because of the data. All the images were 28×28 grayscale, with the item of clothing centered in the frame. It's a good start, but it is a very controlled scenario. To do better at vision, we might need the computer to learn features of an image instead of merely the raw pixels.

We can do that with a process called *convolutions*. You'll learn how to define convolutional neural networks to understand the contents of images in the next chapter.

Going Beyond the Basics: Detecting Features in Images

In [Chapter 2](#) you learned how to get started with computer vision by creating a simple neural network that matched the input pixels of the Fashion MNIST dataset to 10 labels, each representing a type (or class) of clothing. And while you created a network that was pretty good at detecting clothing types, there was a clear drawback. Your neural network was trained on small monochrome images that each contained only a single item of clothing, and that item was centered within the image.

To take the model to the next level, you need to be able to detect *features* in images. So, for example, instead of looking merely at the raw pixels in the image, what if we could have a way to filter the images down to constituent elements? Matching those elements, instead of raw pixels, would help us to detect the contents of images more effectively. Consider the Fashion MNIST dataset that we used in the last chapter—when detecting a shoe, the neural network may have been activated by lots of dark pixels clustered at the bottom of the image, which it would see as the sole of the shoe. But when the shoe is no longer centered and filling the frame, this logic doesn't hold.

One method to detect features comes from photography and the image processing methodologies that you might be familiar with. If you've ever used a tool like Photoshop or GIMP to sharpen an image, you're using a mathematical filter that works on the pixels of the image. Another word for these filters is a *convolution*, and by using these in a neural network you will create a *convolutional neural network* (CNN).

In this chapter you'll learn about how to use convolutions to detect features in an image. You'll then dig deeper into classifying images based on the features within. We'll explore augmentation of images to get more features and transfer learning to take preexisting features that were learned by others, and then look briefly into optimizing your models using dropouts.

Convolutions

A convolution is simply a filter of weights that are used to multiply a pixel with its neighbors to get a new value for the pixel. For example, consider the ankle boot image from Fashion MNIST and the pixel values for it as shown in **Figure 3-1**.

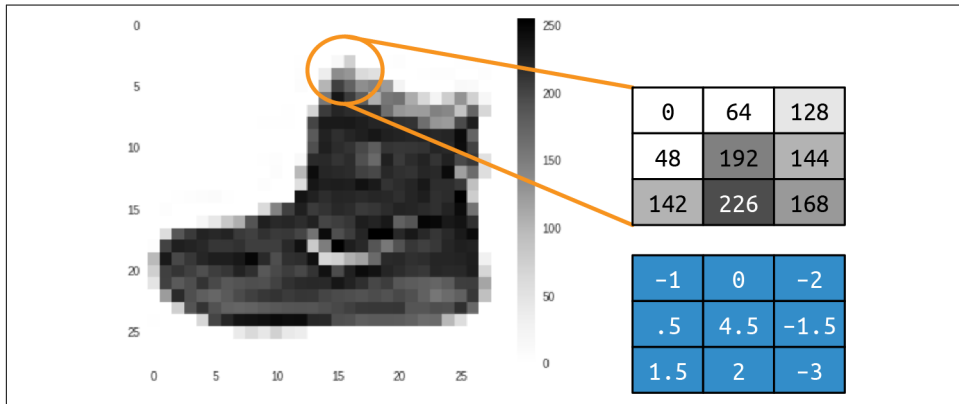


Figure 3-1. Ankle boot with convolution

If we look at the pixel in the middle of the selection we can see that it has the value 192 (recall that Fashion MNIST uses monochrome images with pixel values from 0 to 255). The pixel above and to the left has the value 0, the one immediately above has the value 64, etc.

If we then define a filter in the same 3×3 grid, as shown below the original values, we can transform that pixel by calculating a new value for it. We do this by multiplying the current value of each pixel in the grid by the value in the same position in the filter grid, and summing up the total amount. This total will be the new value for the current pixel. We then repeat this for all pixels in the image.

So in this case, while the current value of the pixel in the center of the selection is 192, the new value after applying the filter will be:

$$\begin{aligned} \text{new_val} = & (-1 * 0) + (0 * 64) + (-2 * 128) + \\ & (.5 * 48) + (4.5 * 192) + (-1.5 * 144) + \\ & (1.5 * 142) + (2 * 226) + (-3 * 168) \end{aligned}$$

This equals 577, which will be the new value for that pixel. Repeating this process across every pixel in the image will give us a filtered image.

Let's consider the impact of applying a filter on a more complicated image: the **ascent image** that's built into SciPy for easy testing. This is a 512×512 grayscale image that shows two people climbing a staircase.

Using a filter with negative values on the left, positive values on the right, and zeros in the middle will end up removing most of the information from the image except for vertical lines, as you can see in [Figure 3-2](#).

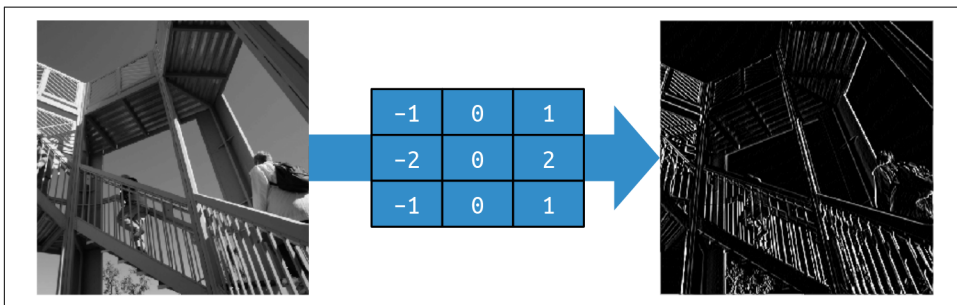


Figure 3-2. Using a filter to get vertical lines

Similarly, a small change to the filter can emphasize the horizontal lines, as shown in [Figure 3-3](#).

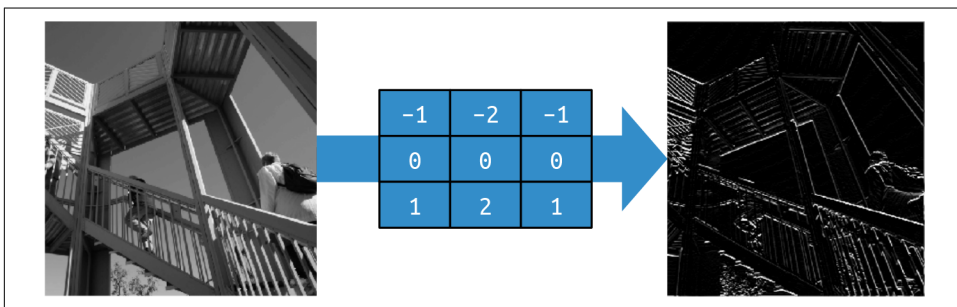


Figure 3-3. Using a filter to get horizontal lines

These examples also show that the amount of information in the image is reduced, so we can potentially *learn* a set of filters that reduce the image to features, and those features can be matched to labels as before. Previously, we *learned* parameters that were used in neurons to match inputs to outputs. Similarly, the best filters to match inputs to outputs can be learned over time.

When combined with *pooling*, we can reduce the amount of information in the image while maintaining the features. We'll explore that next.

Pooling

Pooling is the process of eliminating pixels in your image while maintaining the semantics of the content within the image. It's best explained visually. [Figure 3-4](#) shows the concept of a *max* pooling.

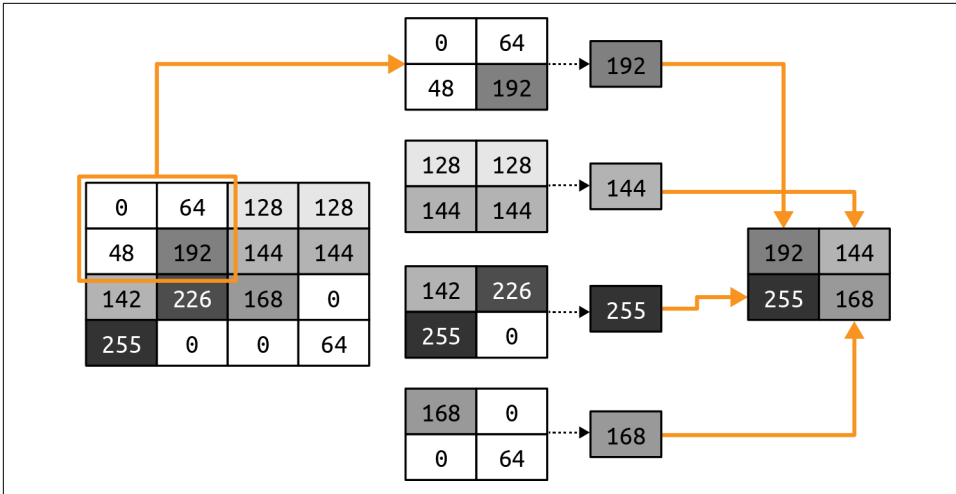


Figure 3-4. Demonstrating max pooling

In this case, consider the box on the left to be the pixels in a monochrome image. We then group them into 2×2 arrays, so in this case the 16 pixels are grouped into four 2×2 arrays. These are called *pools*.

We then select the *maximum* value in each of the groups, and reassemble those into a new image. Thus, the pixels on the left are reduced by 75% (from 16 to 4), with the maximum value from each pool making up the new image.

Figure 3-5 shows the version of ascent from Figure 3-2, with the vertical lines enhanced, after max pooling has been applied.

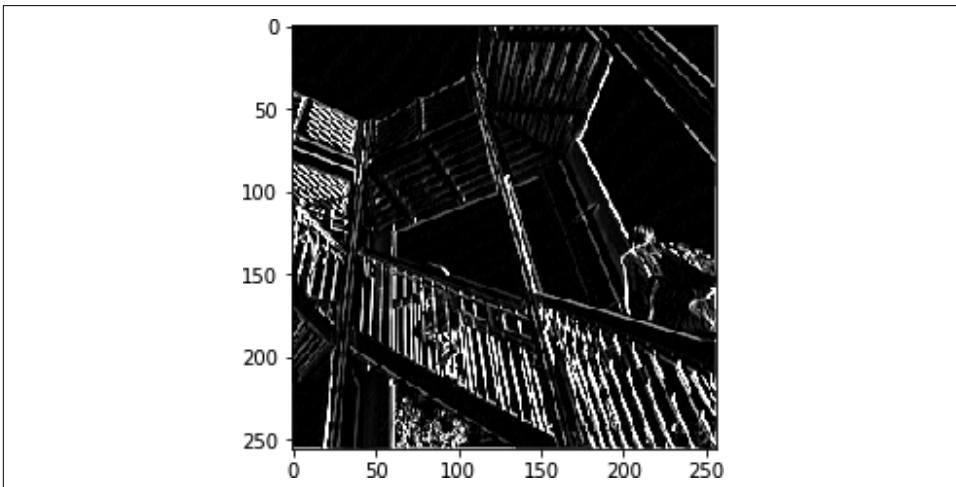


Figure 3-5. Ascent after vertical filter and max pooling

Note how the filtered features have not just been maintained, but further enhanced. Also, the image size has changed from 512×512 to 256×256 —a quarter of the original size.



There are other approaches to pooling, such as *min* pooling, which takes the smallest pixel value from the pool, and *average* pooling, which takes the overall average value.

Implementing Convolutional Neural Networks

In [Chapter 2](#) you created a neural network that recognized fashion images. For convenience, here's the complete code:

```
import tensorflow as tf
data = tf.keras.datasets.fashion_mnist

(training_images, training_labels), (test_images, test_labels) = data.load_data()

training_images = training_images / 255.0
test_images = test_images / 255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(training_images, training_labels, epochs=5)
```

To convert this to a convolutional neural network, we simply use convolutional layers in our model definition. We'll also add pooling layers.

To implement a convolutional layer, you'll use the `tf.keras.layers.Conv2D` type. This accepts as parameters the number of convolutions to use in the layer, the size of the convolutions, the activation function, etc.

For example, here's a convolutional layer used as the input layer to a neural network:

```
tf.keras.layers.Conv2D(64, (3, 3), activation='relu',
                       input_shape=(28, 28, 1)),
```

In this case, we want the layer to learn 64 convolutions. It will randomly initialize these, and over time will learn the filter values that work best to match the input values to their labels. The `(3, 3)` indicates the size of the filter. Earlier I showed 3×3

filters, and that's what we are specifying here. This is the most common size of filter; you can change it as you see fit, but you'll typically see an odd number of axes like 5×5 or 7×7 because of how filters remove pixels from the borders of the image, as you'll see later.

The `activation` and `input_shape` parameters are the same as before. As we're using Fashion MNIST in this example, the shape is still 28×28 . Do note, however, that because Conv2D layers are designed for multicolor images, we're specifying the third dimension as 1, so our input shape is $28 \times 28 \times 1$. Color images will typically have a 3 as the third parameter as they are stored as values of R, G, and B.

Here's how to use a pooling layer in the neural network. You'll typically do this immediately after the convolutional layer:

```
tf.keras.layers.MaxPooling2D(2, 2),
```

In the example in [Figure 3-4](#), we split the image into 2×2 pools and picked the maximum value in each. This operation could have been parameterized to define the pool size. Those are the parameters that you can see here—the (2, 2) indicates that our pools are 2×2 .

Now let's explore the full code for Fashion MNIST with a CNN:

```
import tensorflow as tf
data = tf.keras.datasets.fashion_mnist

(training_images, training_labels), (test_images, test_labels) = data.load_data()

training_images = training_images.reshape(60000, 28, 28, 1)
training_images = training_images / 255.0
test_images = test_images.reshape(10000, 28, 28, 1)
test_images = test_images / 255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu',
                           input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(training_images, training_labels, epochs=50)

model.evaluate(test_images, test_labels)
```

```

classifications = model.predict(test_images)
print(classifications[0])
print(test_labels[0])

```

There are a few things to note here. Remember earlier when I said that the input shape for the images had to match what a Conv2D layer would expect, and we updated it to be a $28 \times 28 \times 1$ image? The data also had to be reshaped accordingly. 28×28 is the number of pixels in the image, and 1 is the number of color channels. You'll typically find that this is 1 for a grayscale image or 3 for a color image, where there are three channels (red, green, and blue), with the number indicating the intensity of that color.

So, prior to normalizing the images, we also reshape each array to have that extra dimension. The following code changes our training dataset from 60,000 images, each 28×28 (and thus a $60,000 \times 28 \times 28$ array), to 60,000 images, each $28 \times 28 \times 1$:

```

training_images = training_images.reshape(60000, 28, 28, 1)

```

We then do the same thing with the test dataset.

Also note that in the original deep neural network (DNN) we ran the input through a Flatten layer prior to feeding it into the first Dense layer. We've lost that in the input layer here—instead, we just specify the input shape. Note that prior to the Dense layer, after convolutions and pooling, the data will be flattened.

Training this network on the same data for the same 50 epochs as the network shown in [Chapter 2](#), we can see a nice increase in accuracy. While the previous example reached 89% accuracy on the test set in 50 epochs, this one will hit 99% in around half that many—24 or 25 epochs. So we can see that adding convolutions to the neural network is definitely increasing its ability to classify images. Let's next take a look at the journey an image takes through the network so we can get a little bit more of an understanding of why this works.

Exploring the Convolutional Network

You can inspect your model using the `model.summary` command. When you run it on the Fashion MNIST convolutional network we've been working on you'll see something like this:

```

Model: "sequential"

```

Layer (type)	Output Shape	Param #
=====		
conv2d (Conv2D)	(None, 26, 26, 64)	640

max_pooling2d (MaxPooling2D)	(None, 13, 13, 64)	0

```
conv2d_1 (Conv2D)          (None, 11, 11, 64) 36928
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64) 0
flatten (Flatten)          (None, 1600) 0
dense (Dense)              (None, 128) 204928
dense_1 (Dense)            (None, 10) 1290
=====
Total params: 243,786
Trainable params: 243,786
Non-trainable params: 0
```

Let's first take a look at the Output Shape column to understand what is going on here. Our first layer will have 28×28 images, and apply 64 filters to them. But because our filter is 3×3 , a 1-pixel border around the image will be lost, reducing our overall information to 26×26 pixels. Consider [Figure 3-6](#). If we take each of the boxes as a pixel in the image, the first possible filter we can do starts at the second row and the second column. The same would happen on the right side and at the bottom of the diagram.

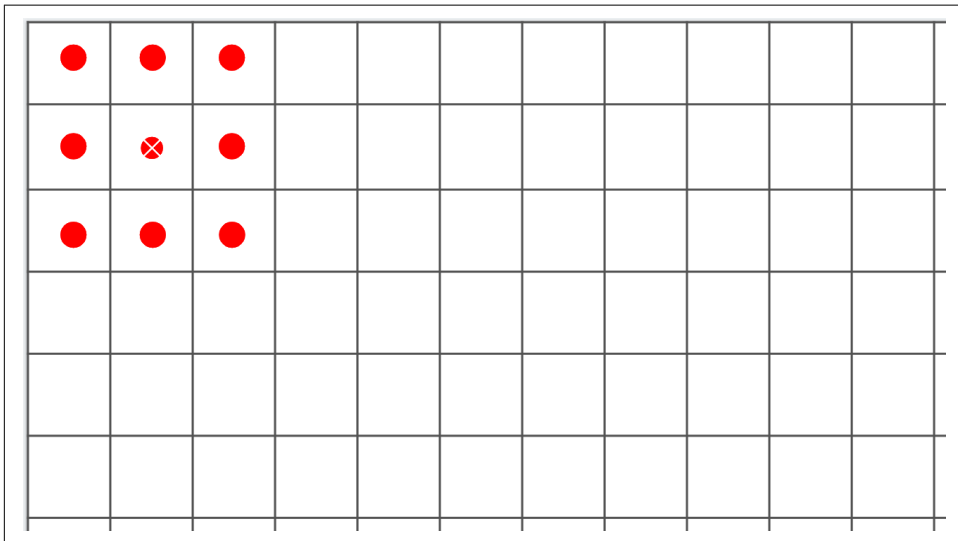


Figure 3-6. Losing pixels when running a filter

Thus, an image that is $A \times B$ pixels in shape when run through a 3×3 filter will become $(A-2) \times (B-2)$ pixels in shape. Similarly, a 5×5 filter would make it $(A-4) \times (B-4)$, and so on. As we're using a 28×28 image and a 3×3 filter, our output will now be 26×26 .

After that the pooling layer is 2×2 , so the size of the image will halve on each axis, and it will then become (13×13) . The next convolutional layer will reduce this further to 11×11 , and the next pooling, rounding down, will make the image 5×5 .

So, by the time the image has gone through two convolutional layers, the result will be many 5×5 images. How many? We can see that in the Param # (parameters) column.

Each convolution is a 3×3 filter, plus a bias. Remember earlier with our dense layers, each layer was $Y = mX + c$, where m was our parameter (aka weight) and c was our bias? This is very similar, except that because the filter is 3×3 there are 9 parameters to learn. Given that we have 64 convolutions defined, we'll have 640 overall parameters (each convolution has 9 parameters plus a bias, for a total of 10, and there are 64 of them).

The MaxPooling layers don't learn anything, they just reduce the image, so there are no learned parameters there—hence 0 being reported.

The next convolutional layer has 64 filters, but each of these is multiplied across the *previous* 64 filters, each with 9 parameters. We have a bias on each of the new 64 filters, so our number of parameters should be $(64 \times (64 \times 9)) + 64$, which gives us 36,928 parameters the network needs to learn.

If this is confusing, try changing the number of convolutions in the first layer to something—for example, 10. You'll see the number of parameters in the second layer becomes 5,824, which is $(64 \times (10 \times 9)) + 64$.

By the time we get through the second convolution, our images are 5×5 , and we have 64 of them. If we multiply this out we now have 1,600 values, which we'll feed into a dense layer of 128 neurons. Each neuron has a weight and a bias, and we have 128 of them, so the number of parameters the network will learn is $((5 \times 5 \times 64) \times 128) + 128$, giving us 204,928 parameters.

Our final dense layer of 10 neurons takes in the output of the previous 128, so the number of parameters learned will be $(128 \times 10) + 10$, which is 1,290.

The total number of parameters is then the sum of all of these: 243,786.

Training this network requires us to learn the best set of these 243,786 parameters to match the input images to their labels. It's a slower process because there are more parameters, but as we can see from the results, it also builds a more accurate model!

Of course, with this dataset we still have the limitation that the images are 28×28 , monochrome, and centered. Next we'll take a look at using convolutions to explore a more complex dataset comprising color pictures of horses and humans, and we'll try to determine if an image contains one or the other. In this case, the subject won't

always be centered in the image like with Fashion MNIST, so we'll have to rely on convolutions to spot distinguishing features.

Building a CNN to Distinguish Between Horses and Humans

In this section we'll explore a more complex scenario than the Fashion MNIST classifier. We'll extend what we've learned about convolutions and convolutional neural networks to try to classify the contents of images where the location of a feature isn't always in the same place. I've created the Horses or Humans dataset for this purpose.

The Horses or Humans Dataset

The dataset for [this section](#) contains over a thousand 300×300 -pixel images, approximately half each of horses and humans, rendered in different poses. You can see some examples in [Figure 3-7](#).

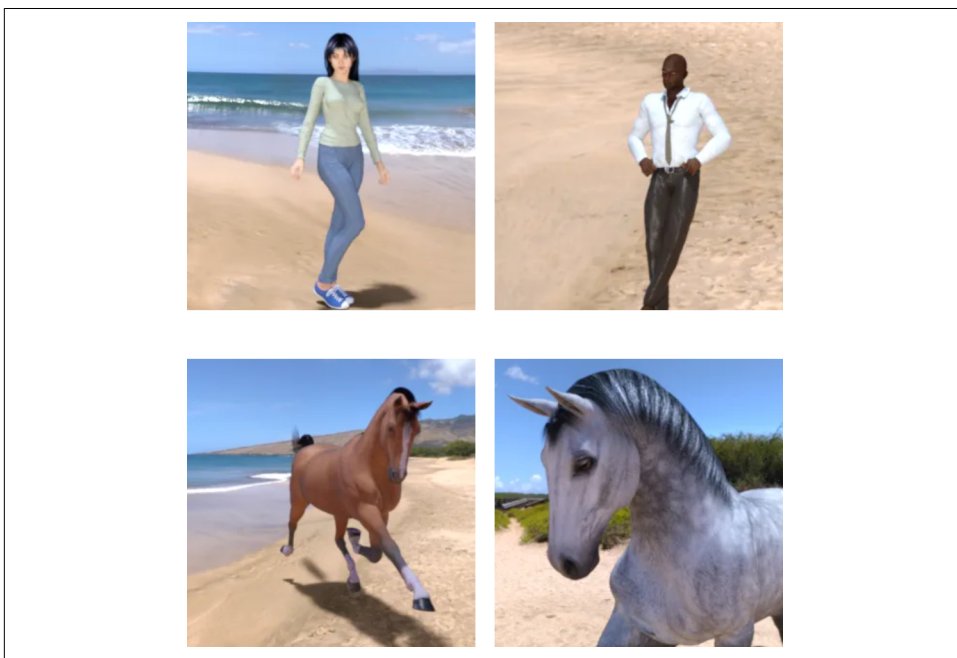


Figure 3-7. Horses and humans

As you can see, the subjects have different orientations and poses and the image composition varies. Consider the two horses, for example—their heads are oriented differently, and one is zoomed out showing the complete animal while the other is

zoomed in, showing just the head and part of the body. Similarly, the humans are lit differently, have different skin tones, and are posed differently. The man has his hands on his hips, while the woman has hers outstretched. The images also contain backgrounds such as trees and beaches, so a classifier will have to determine which parts of the image are the important features that determine what makes a horse a horse and a human a human, without being affected by the background.

While the previous examples of predicting $Y = 2X - 1$ or classifying small monochrome images of clothing *might* have been possible with traditional coding, it's clear that this is far more difficult, and you are crossing the line into where machine learning is essential to solve a problem.

An interesting side note is that these images are all computer-generated. The theory is that features spotted in a CGI image of a horse should apply to a real image. You'll see how well this works later in this chapter.

The Keras ImageDataGenerator

The Fashion MNIST dataset that you've been using up to this point comes with labels. Every image file has an associated file with the label details. Many image-based datasets do not have this, and Horses or Humans is no exception. Instead of labels, the images are sorted into subdirectories of each type. With Keras in TensorFlow, a tool called the ImageDataGenerator can use this structure to *automatically* assign labels to images.

To use the ImageDataGenerator, you simply ensure that your directory structure has a set of named subdirectories, with each subdirectory being a label. For example, the Horses or Humans dataset is available as a set of ZIP files, one with the training data (1,000+ images) and another with the validation data (256 images). When you download and unpack them into a local directory for training and validation, ensure that they are in a file structure like the one in [Figure 3-8](#).

Here's the code to get the training data and extract it into the appropriately named subdirectories, as shown in this figure:

```
import urllib.request
import zipfile

url = "https://storage.googleapis.com/laurencemoroney-blog.appspot.com/"
                                     horse-or-human.zip"

file_name = "horse-or-human.zip"
training_dir = 'horse-or-human/training/'
urllib.request.urlretrieve(url, file_name)

zip_ref = zipfile.ZipFile(file_name, 'r')
zip_ref.extractall(training_dir)
zip_ref.close()
```

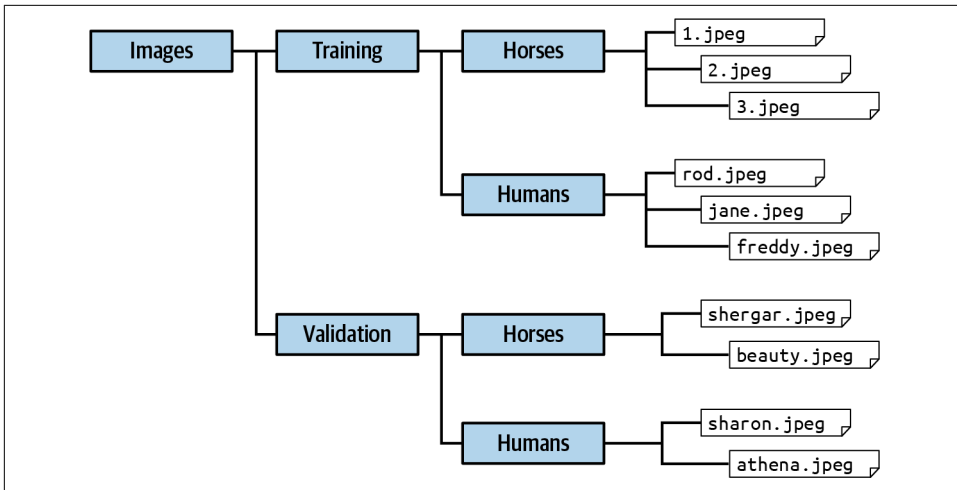


Figure 3-8. Ensuring that images are in named subdirectories

Here's the code to get the training data and extract it into the appropriately named subdirectories, as shown in this figure:

```
import urllib.request
import zipfile

url = "https://storage.googleapis.com/laurencemoroney-blog.appspot.com/
                                         horse-or-human.zip"

file_name = "horse-or-human.zip"
training_dir = 'horse-or-human/training/'
urllib.request.urlretrieve(url, file_name)

zip_ref = zipfile.ZipFile(file_name, 'r')
zip_ref.extractall(training_dir)
zip_ref.close()
```

This simply downloads the ZIP of the training data and unzips it into a directory at *horse-or-human/training* (we'll deal with downloading the validation data shortly). This is the parent directory that will contain subdirectories for the image types.

To use the `ImageDataGenerator` we now simply use the following code:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# All images will be rescaled by 1./255
train_datagen = ImageDataGenerator(rescale=1/255)

train_generator = train_datagen.flow_from_directory(
    training_dir,
    target_size=(300, 300),
    class_mode='binary'
)
```

We first create an instance of an `ImageDataGenerator` called `train_datagen`. We then specify that this will generate images for the training process by flowing them from a directory. The directory is *training_dir*, as specified earlier. We also indicate some hyperparameters about the data, such as the target size—in this case the images are 300×300 , and the class mode is binary. The mode is usually binary if there are just two types of images (as in this case) or categorical if there are more than two.

CNN Architecture for Horses or Humans

There are several major differences between this dataset and the Fashion MNIST one that you have to take into account when designing an architecture for classifying the images. First, the images are much larger— 300×300 pixels—so more layers may be needed. Second, the images are full color, not grayscale, so each image will have three channels instead of one. Third, there are only two image types, so we have a binary classifier that can be implemented using just a single output neuron, where it approaches 0 for one class and 1 for the other. Keep these considerations in mind when exploring this architecture:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(16, (3,3), activation='relu',
                           input_shape=(300, 300, 3)),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

There are a number of things to note here. First of all, this is the very first layer. We're defining 16 filters, each 3×3 , but the input shape of the image is $(300, 300, 3)$. Remember that this is because our input image is 300×300 and it's in color, so there are three channels, instead of just one for the monochrome Fashion MNIST dataset we were using earlier.

At the other end, notice that there's only one neuron in the output layer. This is because we're using a binary classifier, and we can get a binary classification with just a single neuron if we activate it with a sigmoid function. The purpose of the sigmoid function is to drive one set of values toward 0 and the other toward 1, which is perfect for binary classification.

Next, notice how we stack several more convolutional layers. We do this because our image source is quite large, and we want, over time, to have many smaller images, each with features highlighted. If we take a look at the results of `model.summary` we'll see this in action:

=====		
conv2d (Conv2D)	(None, 298, 298, 16)	448
<hr/>		
max_pooling2d (MaxPooling2D)	(None, 149, 149, 16)	0
<hr/>		
conv2d_1 (Conv2D)	(None, 147, 147, 32)	4640
<hr/>		
max_pooling2d_1 (MaxPooling2D)	(None, 73, 73, 32)	0
<hr/>		
conv2d_2 (Conv2D)	(None, 71, 71, 64)	18496
<hr/>		
max_pooling2d_2 (MaxPooling2D)	(None, 35, 35, 64)	0
<hr/>		
conv2d_3 (Conv2D)	(None, 33, 33, 64)	36928
<hr/>		
max_pooling2d_3 (MaxPooling2D)	(None, 16, 16, 64)	0
<hr/>		
conv2d_4 (Conv2D)	(None, 14, 14, 64)	36928
<hr/>		
max_pooling2d_4 (MaxPooling2D)	(None, 7, 7, 64)	0
<hr/>		
flatten (Flatten)	(None, 3136)	0
<hr/>		
dense (Dense)	(None, 512)	1606144
<hr/>		
dense_1 (Dense)	(None, 1)	513
<hr/>		
=====		
Total params: 1,704,097		
Trainable params: 1,704,097		
Non-trainable params: 0		
<hr/>		

Note how, by the time the data has gone through all the convolutional and pooling layers, it ends up as 7×7 items. The theory is that these will be activated feature maps that are relatively simple, containing just 49 pixels. These feature maps can then be passed to the dense neural network to match them to the appropriate labels.

This, of course, leads us to have many more parameters than the previous network, so it will be slower to train. With this architecture, we're going to learn 1.7 million parameters.

To train the network, we'll have to compile it with a loss function and an optimizer. In this case the loss function can be binary cross entropy loss function binary cross entropy, because there are only two classes, and as the name suggests this is a loss function that is designed for that scenario. And we can try a new optimizer, *root mean square propagation* (RMSprop), that takes a learning rate (lr) parameter that allows us to tweak the learning. Here's the code:

```
model.compile(loss='binary_crossentropy',
              optimizer=RMSprop(lr=0.001),
              metrics=['accuracy'])
```

We train by using `fit_generator` and passing it the `training_generator` we created earlier:

```
history = model.fit_generator(
    train_generator,
    epochs=15
)
```

This sample will work in Colab, but if you want to run it on your own machine, please ensure that the Pillow libraries are installed using `pip install pillow`.

Note that with TensorFlow Keras, you can use `model.fit` to fit your training data to your training labels. When using a generator, older versions required you to use `model.fit_generator` instead. Later versions of TensorFlow will allow you to use either.

Over just 15 epochs, this architecture gives us a very impressive 95%+ accuracy on the training set. Of course, this is just with the training data, and isn't an indication of performance on data that the network hasn't previously seen.

Next we'll look at adding the validation set using a generator and measuring its performance to give us a good indication of how this model might perform in real life.

Adding Validation to the Horses or Humans Dataset

To add validation, you'll need a validation dataset that's separate from the training one. In some cases you'll get a master dataset that you have to split yourself, but in the case of Horses or Humans, there's a separate validation set that you can download.



You may be wondering why we're talking about a validation dataset here, rather than a test dataset, and whether they're the same thing. For simple models like the ones developed in the previous chapters, it's often sufficient to split the dataset into two parts, one for training and one for testing. But for more complex models like the one we're building here, you'll want to create separate validation and test sets. What's the difference? *Training* data is the data that is used to teach the network how the data and labels fit together. *Validation* data is used to see how the network is doing with previously unseen data *while* you are training—i.e., it isn't used for fitting data to labels, but to inspect how well the fitting is going. *Test* data is used *after* training to see how the network does with data it has never previously seen. Some datasets come with a three-way split, and in other cases you'll want to separate the test set into two parts for validation and testing. Here, you'll download some additional images for testing the model.

You can use very similar code to that used for the training images to download the validation set and unzip it into a different directory:

```
validation_url = "https://storage.googleapis.com/laurencemoroney-blog.appspot.com  
                /validation-horse-or-human.zip"  
  
validation_file_name = "validation-horse-or-human.zip"  
validation_dir = 'horse-or-human/validation/'  
urllib.request.urlretrieve(validation_url, validation_file_name)  
  
zip_ref = zipfile.ZipFile(validation_file_name, 'r')  
zip_ref.extractall(validation_dir)  
zip_ref.close()
```

Once you have the validation data, you can set up another `ImageDataGenerator` to manage these images:

```
validation_datagen = ImageDataGenerator(rescale=1/255)  
  
validation_generator = train_datagen.flow_from_directory(  
    validation_dir,  
    target_size=(300, 300),  
    class_mode='binary'  
)
```

To have TensorFlow perform the validation for you, you simply update your `model.fit_generator` method to indicate that you want to use the validation data to test the model epoch by epoch. You do this by using the `validation_data` parameter and passing it the validation generator you just constructed:

```
history = model.fit_generator(  
    train_generator,  
    epochs=15,
```

```
validation_data=validation_generator  
)
```

After training for 15 epochs, you should see that your model is 99%+ accurate on the training set, but only about 88% on the validation set. This is an indication that the model is overfitting, as we saw in the previous chapter.

Still, the performance isn't bad considering how few images it was trained on, and how diverse those images were. You're beginning to hit a wall caused by lack of data, but there are some techniques that you can use to improve your model's performance. We'll explore them later in this chapter, but before that let's take a look at how to *use* this model.

Testing Horse or Human Images

It's all very well to be able to build a model, but of course you want to try it out. A major frustration of mine when I was starting my AI journey was that I could find lots of code that showed me how to build models, and charts of how those models were performing, but very rarely was there code to help me kick the tires of the model myself to try it out. I'll try to avoid that in this book!

Testing the model is perhaps easiest using Colab. I've provided a Horses or Humans notebook on GitHub that you can open directly in [Colab](#).

Once you've trained the model, you'll see a section called "Running the Model." Before running it, find a few pictures of horses or humans online and download them to your computer. [Pixabay.com](#) is a really good site to check out for royalty-free images. It's a good idea to get your test images together first, because the node can time out while you're searching.

Figure 3-9 shows a few pictures of horses and humans that I downloaded from Pixabay to test the model.

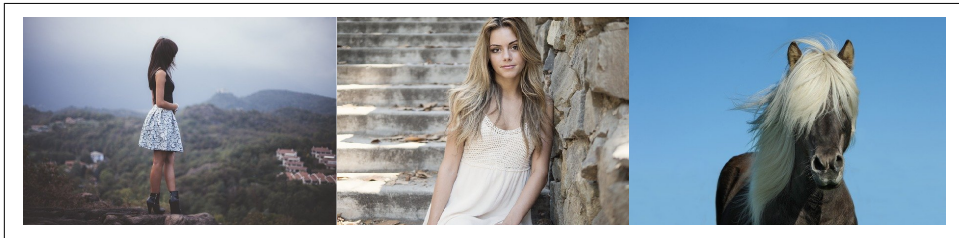


Figure 3-9. Test images

When they were uploaded, as you can see in **Figure 3-10**, the model correctly classified the first image as a human and the third image as a horse, but the middle image, despite being obviously a human, was incorrectly classified as a horse!


```
import numpy as np
from google.colab import files
from keras.preprocessing import image

uploaded = files.upload()

for fn in uploaded.keys():

    # predicting images
    path = '/content/' + fn
    img = image.load_img(path, target_size=(300, 300))
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)

    image_tensor = np.vstack([x])
    classes = model.predict(image_tensor)
    print(classes)
    print(classes[0])
    if classes[0]>0.5:
        print(fn + " is a human")
    else:
        print(fn + " is a horse")
```

Choose Files 3 files

- **model-993907_640.jpg**(image/jpeg) - 36254 bytes, last modified: 2/15/2020 - 100% done
- **beautiful-1274056_640.jpg**(image/jpeg) - 80225 bytes, last modified: 2/15/2020 - 100% done
- **horse-1330690_640.jpg**(image/jpeg) - 33396 bytes, last modified: 2/15/2020 - 100% done

Saving model-993907_640.jpg to model-993907_640 (3).jpg
Saving beautiful-1274056_640.jpg to beautiful-1274056_640 (1).jpg
Saving horse-1330690_640.jpg to horse-1330690_640 (2).jpg

```
[[1.]]
[1.]
model-993907_640.jpg is a human
[[0.]]
[0.]
beautiful-1274056_640.jpg is a horse
[[0.]]
[0.]
horse-1330690_640.jpg is a horse
```

Figure 3-10. Executing the model

You can also upload multiple images simultaneously and have the model make predictions for all of them. You may notice that it tends to overfit toward horses. If the human isn't fully posed—i.e., you can't see their full body—it can skew toward horses. That's what happened in this case. The first human model is fully posed and the image resembles many of the poses in the dataset, so it was able to classify her correctly. The second model was facing the camera, but only her upper half is in the image. There was no training data that looked like that, so the model couldn't correctly identify her.

Let's now explore the code to see what it's doing. Perhaps the most important part is this chunk:

```
img = image.load_img(path, target_size=(300, 300))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
```

Here, we are loading the image from the path that Colab wrote it to. Note that we specify the target size to be 300×300 . The images being uploaded can be any shape, but if we are going to feed them into the model, they *must* be 300×300 , because that's what the model was trained to recognize. So, the first line of code loads the image and *resizes* it to 300×300 .

The next line of code converts the image into a 2D array. The model, however, expects a 3D array, as indicated by the `input_shape` in the model architecture. Fortunately, Numpy provides an `expand_dims` method that handles this and allows us to easily add a new dimension to the array.

Now that we have our image in a 3D array, we just want to make sure that it's stacked vertically so that it is in the same shape as the training data:

```
image_tensor = np.vstack([x])
```

With our image in the right format, it's easy to do the classification:

```
classes = model.predict(image_tensor)
```

The model returns an array containing the classifications. Because there's only one classification in this case, it's effectively an array containing an array. You can see this in [Figure 3-10](#), where for the first (human) model it looks like `[[1.]]`.

So now it's simply a matter of inspecting the value of the first element in that array. If it's greater than 0.5, we're looking at a human:

```
if classes[0]>0.5:
    print(fn + " is a human")
else:
    print(fn + " is a horse")
```

There are a few important points to consider here. First, even though the network was trained on synthetic, computer-generated imagery, it performs quite well at spotting horses or humans in real photographs. This is a potential boon in that you may not need thousands of photographs to train a model, and can do it relatively cheaply with CGI.

But this dataset also demonstrates a fundamental issue you will face. Your training set cannot hope to represent *every* possible scenario your model might face in the wild, and thus the model will always have some level of overspecialization toward the training set. A clear and simple example of that was shown here, where the human in the center of [Figure 3-9](#) was miscategorized. The training set didn't include a human in

that pose, and thus the model didn't "learn" that a human could look like that. As a result, there was every chance it might see the figure as a horse, and in this case, it did.

What's the solution? The obvious one is to add more training data, with humans in that particular pose and others that weren't initially represented. That isn't always possible, though. Fortunately, there's a neat trick in TensorFlow that you can use to virtually extend your dataset—it's called *image augmentation*, and we'll explore that next.

Image Augmentation

In the previous section, you built a horse-or-human classifier model that was trained on a relatively small dataset. As a result, you soon began to hit problems classifying some previously unseen images, such as the miscategorization of a woman with a horse because the training set didn't include any images of people in that pose.

One way to deal with such problems is with image augmentation. The idea behind this technique is that, as TensorFlow is loading your data, it can create additional new data by amending what it has using a number of transforms. For example, take a look at [Figure 3-11](#). While there is nothing in the dataset that looks like the woman on the right, the image on the left is somewhat similar.



Figure 3-11. Dataset similarities

So if you could, for example, zoom into the image on the left as you are training, as shown in [Figure 3-12](#), you would increase the chances of the model being able to correctly classify the image on the right as a person.



Figure 3-12. Zooming in on the training set data

In a similar way, you can broaden the training set with a variety of other transformations, including:

- Rotation
- Shifting horizontally
- Shifting vertically
- Shearing
- Zooming
- Flipping

Because you’ve been using the `ImageDataGenerator` to load the images, you’ve seen it do a transform already—when it normalized the images like this:

```
train_datagen = ImageDataGenerator(rescale=1/255)
```

The other transforms are easily available within the `ImageDataGenerator` too, so, for example, you could do something like this:

```
train_datagen = ImageDataGenerator(  
    rescale=1./255,  
    rotation_range=40,  
    width_shift_range=0.2,  
    height_shift_range=0.2,  
    shear_range=0.2,  
    zoom_range=0.2,  
    horizontal_flip=True,  
    fill_mode='nearest'  
)
```

Here, as well as rescaling the image to normalize it, you're also doing the following:

- Rotating each image randomly up to 40 degrees left or right
- Translating the image up to 20% vertically or horizontally
- Shearing the image by up to 20%
- Zooming the image by up to 20%
- Randomly flipping the image horizontally or vertically
- Filling in any missing pixels after a move or shear with nearest neighbors

When you retrain with these parameters, one of the first things you'll notice is that training takes longer because of all the image processing. Also, your model's accuracy may not be as high as it was previously, because previously it was overfitting to a largely uniform set of data.

In my case, when training with these augmentations my accuracy went down from 99% to 85% after 15 epochs, with validation slightly higher at 89%. (This indicates that the model is *underfitting* slightly, so the parameters could be tweaked a bit.)

What about the image from [Figure 3-9](#) that it misclassified earlier? This time, it gets it right. Thanks to the image augmentations, the training set now has sufficient coverage for the model to understand that this particular image is a human too (see [Figure 3-13](#)). This is just a single data point, and may not be representative of the results for real data, but it's a small step in the right direction.

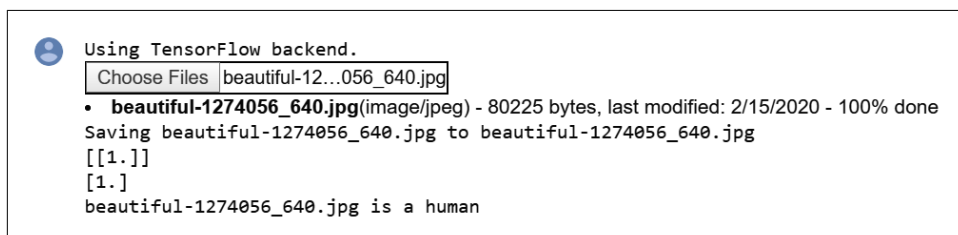


Figure 3-13. The zoomed woman is now correctly classified

As you can see, even with a relatively small dataset like Horses or Humans you can start to build a pretty decent classifier. With larger datasets you could take this further. Another technique to improve the model is to use features that were already learned elsewhere. Many researchers with massive resources (millions of images) and huge models that have been trained on thousands of classes have shared their models, and using a concept called *transfer learning* you can use the features those models learned and apply them to your data. We'll explore that next!

Transfer Learning

As we've already seen in this chapter, the use of convolutions to extract features can be a powerful tool for identifying the contents of an image. The resulting feature maps can then be fed into the dense layers of a neural network to match them to the labels and give us a more accurate way of determining the contents of an image. Using this approach, with a simple, fast-to-train neural network and some image augmentation techniques, we built a model that was 80–90% accurate at distinguishing between a horse and a human when trained on a very small dataset.

But we can improve our model even further using a method called transfer learning. The idea behind transfer learning is simple: instead of learning a set of filters from scratch for our dataset, why not use a set of filters that were learned on a much larger dataset, with many more features than we can “afford” to build from scratch? We can place these in our network and then train a model with our data using the prelearned filters. For example, our Horses or Humans dataset has only two classes. We can use an existing model that was pretrained for one thousand classes, but at some point we'll have to throw away some of the preexisting network and add the layers that will let us have a classifier for two classes.

Figure 3-14 shows what a CNN architecture for a classification task like ours might look like. We have a series of convolutional layers that lead to a dense layer, which in turn leads to an output layer.

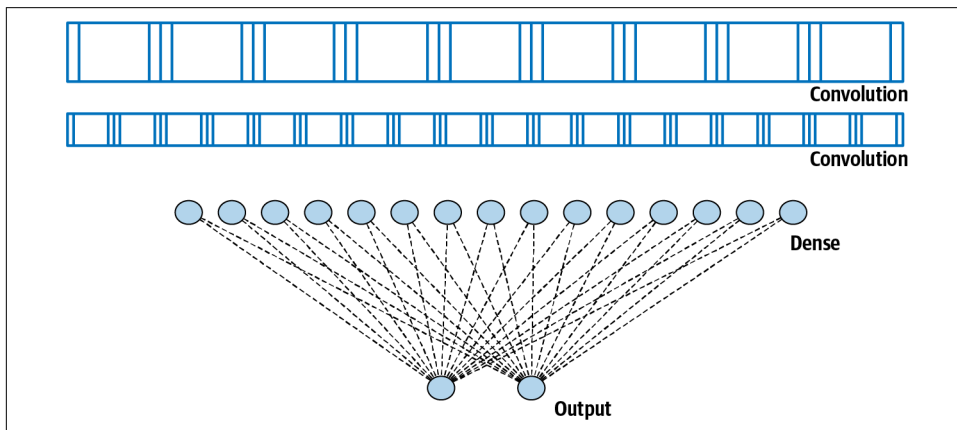


Figure 3-14. A convolutional neural network architecture

We've seen that we're able to build a pretty good classifier using this architecture. But with transfer learning, what if we could take the prelearned layers from another model, freeze or lock them so that they aren't trainable, and then put them on top of our model, like in Figure 3-15?

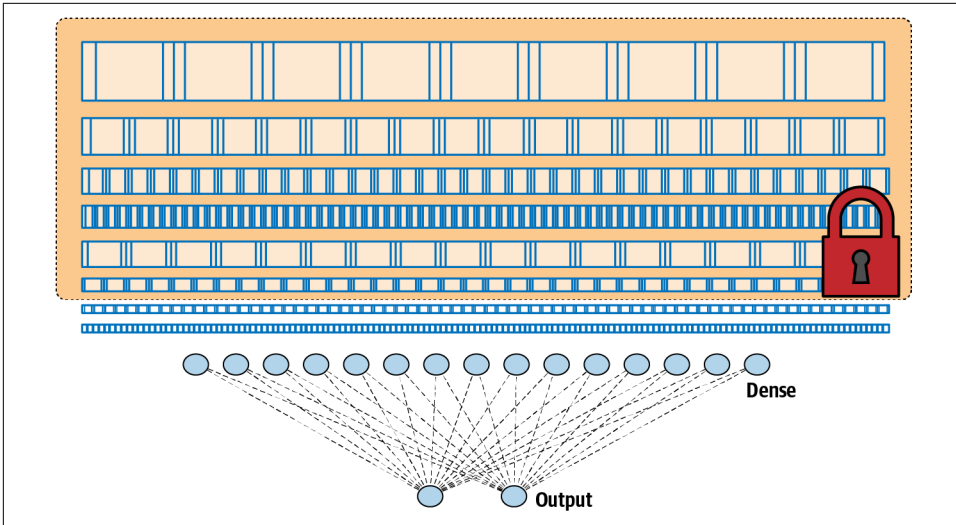


Figure 3-15. Taking layers from another architecture via transfer learning

When we consider that, once they've been trained, all these layers are just a set of numbers indicating the filter values, weights, and biases along with a known architecture (number of filters per layer, size of filter, etc.), the idea of reusing them is pretty straightforward.

Let's look at how this would appear in code. There are several pretrained models already available from a variety of sources. We'll use version 3 of the popular Inception model from Google, which is trained on more than a million images from a database called ImageNet. It has dozens of layers and can classify images into one thousand categories. A saved model is available containing the pretrained weights. To use this, we simply download the weights, create an instance of the Inception V3 architecture, and then load the weights into this architecture like this:

```
from tensorflow.keras.applications.inception_v3 import InceptionV3

weights_url = "https://storage.googleapis.com/mledu-
datasets/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5"

weights_file = "inception_v3.h5"
urlretrieve(weights_url, weights_file)

pre_trained_model = InceptionV3(input_shape=(150, 150, 3),
                                include_top=False,
                                weights=None)

pre_trained_model.load_weights(weights_file)
```

Now we have a full Inception model that's pretrained. If you want to inspect its architecture, you can do so with:

```
pre_trained_model.summary()
```

Be warned—it's huge! Still, take a look through it to see the layers and their names. I like to use the one called `mixed7` because its output is nice and small— 7×7 images—but feel free to experiment with others.

Next, we'll freeze the entire network from retraining and then set a variable to point at `mixed7`'s output as where we want to crop the network up to. We can do that with this code:

```
for layer in pre_trained_model.layers:
    layer.trainable = False

last_layer = pre_trained_model.get_layer('mixed7')
print('last layer output shape: ', last_layer.output_shape)
last_output = last_layer.output
```

Note that we print the output shape of the last layer, and you'll see that we're getting 7×7 images at this point. This indicates that by the time the images have been fed through to `mixed7`, the output images from the filters are 7×7 in size, so they're pretty easy to manage. Again, you don't have to choose that specific layer; you're welcome to experiment with others.

Let's now see how to add our dense layers underneath this:

```
# Flatten the output layer to 1 dimension
x = layers.Flatten()(last_output)
# Add a fully connected layer with 1,024 hidden units and ReLU activation
x = layers.Dense(1024, activation='relu')(x)
# Add a final sigmoid layer for classification
x = layers.Dense(1, activation='sigmoid')(x)
```

It's as simple as creating a flattened set of layers from the last output, because we'll be feeding the results into a dense layer. We then add a dense layer of 1,024 neurons, and a dense layer with 1 neuron for our output.

Now we can define our model simply by saying it's our pretrained model's input followed by the `x` we just defined. We then compile it in the usual way:

```
model = Model(pre_trained_model.input, x)

model.compile(optimizer=RMSprop(lr=0.0001),
              loss='binary_crossentropy',
              metrics=['acc'])
```

Training the model on this architecture over 40 epochs gave an accuracy of 99%+, with a validation accuracy of 96%+ (see [Figure 3-16](#)).


```

34/34 [=====] - 11s 210ms/step - loss: 0.0149 - acc: 0.9971 - val_loss: 0.2030 - val_acc: 0.9709
Epoch 38/40
52/52 [=====] - 11s 211ms/step - loss: 0.0088 - acc: 0.9981 - val_loss: 0.2245 - val_acc: 0.9727
Epoch 39/40
52/52 [=====] - 11s 210ms/step - loss: 0.0088 - acc: 0.9971 - val_loss: 0.2072 - val_acc: 0.9727
Epoch 40/40
52/52 [=====] - 11s 219ms/step - loss: 0.0118 - acc: 0.9971 - val_loss: 0.6150 - val_acc: 0.9609

```

Figure 3-16. Training the horse-or-human classifier with transfer learning

The results here are much better than with our previous model, but you can continue to tweak and improve it. You can also explore how the model will work with a much larger dataset, like the famous **Dogs vs. Cats** from Kaggle. This is an extremely varied dataset consisting of 25,000 images of cats and dogs, often with the subjects somewhat obscured—for example, if they are held by a human.

Using the same algorithm and model design as before you can train a Dogs vs. Cats classifier on Colab, using a GPU at about 3 minutes per epoch. For 20 epochs, this equates to about 1 hour of training.

When tested with very complex pictures like those in **Figure 3-17**, this classifier got them all correct. I chose one picture of a dog with catlike ears, and one with its back turned. Both pictures of cats were nontypical.



Figure 3-17. Unusual dogs and cats that were classified correctly

The cat in the lower-right corner with its eyes closed, ears down, and tongue out while washing its paw gave the results in [Figure 3-18](#) when loaded into the model. You can see that it gave a very low value (4.98×10^{-24}), which shows that the network was *almost* certain it was a cat!

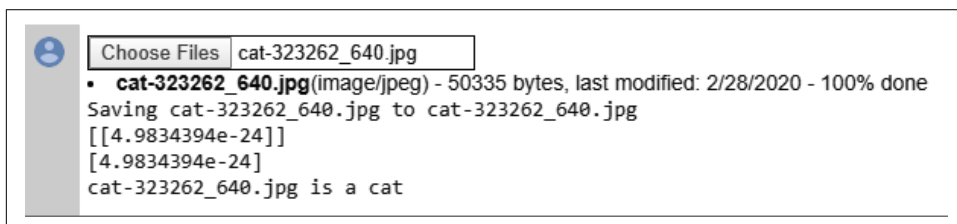


Figure 3-18. Classifying the cat washing its paw

You can find the complete code for the Horses or Humans and Dogs vs. Cats classifiers in the [GitHub repository](#) for this book.

Multiclass Classification

In all of the examples so far you’ve been building *binary* classifiers—ones that choose between two options (horses or humans, cats or dogs). When building multiclass classifiers the models are almost the same, but there are a few important differences. Instead of a single neuron that is sigmoid-activated, or two neurons that are binary-activated, your output layer will now require n neurons, where n is the number of classes you want to classify. You’ll also have to change your loss function to an appropriate one for multiple categories. For example, whereas for the binary classifiers you’ve built so far in this chapter your loss function was binary cross entropy, if you want to extend the model for multiple classes you should instead use categorical cross entropy. If you’re using the `ImageDataGenerator` to provide your images the labeling is done automatically, so multiple categories will work the same as binary ones—the `ImageDataGenerator` will simply label based on the number of subdirectories.

Consider, for example, the game Rock Paper Scissors. If you wanted to train a dataset to recognize the different hand gestures, you’d need to handle three categories. Fortunately, there’s a [simple dataset](#) you can use for this.

There are two downloads: a training set of many diverse hands, with different sizes, shapes, colors, and details such as nail polish; and a testing set of equally diverse hands, none of which are in the training set.

You can see some examples in [Figure 3-19](#).

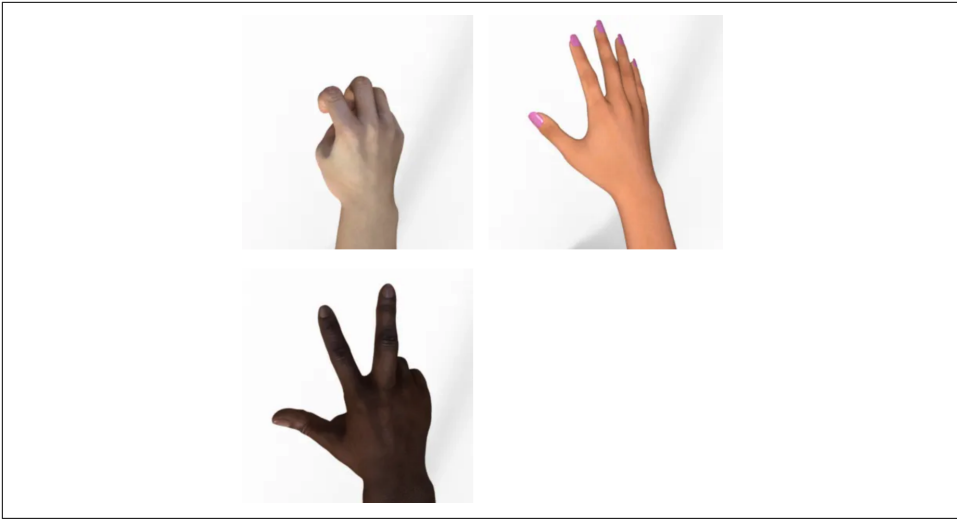


Figure 3-19. Examples of Rock/Paper/Scissors gestures

Using the dataset is simple. Download and unzip it—the sorted subdirectories are already present in the ZIP file—and then use it to initialize an `ImageDataGenerator`:

```
!wget --no-check-certificate \
  https://storage.googleapis.com/laurencemoroney-blog.appspot.com/rps.zip \
  -O /tmp/rps.zip
local_zip = '/tmp/rps.zip'
zip_ref = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall('/tmp/')
zip_ref.close()
TRAINING_DIR = "/tmp/rps/"
training_datagen = ImageDataGenerator(
    rescale = 1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)
```

Note, however, that when you set up the data generator from this, you have to specify that the class mode is categorical in order for the `ImageDataGenerator` to use more than two subdirectories:

```
train_generator = training_datagen.flow_from_directory(
    TRAINING_DIR,
    target_size=(150,150),
    class_mode='categorical'
)
```

When defining your model, while keeping an eye on the input and output layers, you want to ensure that the input matches the shape of the data (in this case 150×150) and that the output matches the number of classes (now three):

```
model = tf.keras.models.Sequential([
    # Note the input shape is the desired size of the image:
    # 150x150 with 3 bytes color
    # This is the first convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu',
                           input_shape=(150, 150, 3)),
    tf.keras.layers.MaxPooling2D(2, 2),
    # The second convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # The third convolution
    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # The fourth convolution
    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # Flatten the results to feed into a DNN
    tf.keras.layers.Flatten(),
    # 512 neuron hidden layer
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(3, activation='softmax')
])
```

Finally, when compiling your model, you want to ensure that it uses a categorical loss function, such as categorical cross entropy. Binary cross entropy will not work with more than two classes:

```
model.compile(loss = 'categorical_crossentropy', optimizer='rmsprop',
              metrics=['accuracy'])
```

Training is then the same as before:

```
history = model.fit(train_generator, epochs=25,
                    validation_data = validation_generator, verbose = 1)
```

Your code for testing predictions will also need to change somewhat. There are now three output neurons, and they will output a value close to 1 for the predicted class, and close to 0 for the other classes. Note that the activation function used is `softmax`, which will ensure that all three predictions will add up to 1. For example, if the model sees something it's really unsure about it might output .4, .4, .2, but if it sees something it's quite sure about you might get .98, .01, .01.

Note also that when using the `ImageDataGenerator`, the classes are loaded in alphabetical order—so while you might expect the output neurons to be in the order of the name of the game, the order in fact will be Paper, Rock, Scissors.

Code to try out predictions in a Colab notebook will look like this. It's very similar to what you saw earlier:

```
import numpy as np
from google.colab import files
from keras.preprocessing import image

uploaded = files.upload()

for fn in uploaded.keys():

    # predicting images
    path = fn
    img = image.load_img(path, target_size=(150, 150))
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)

    images = np.vstack([x])
    classes = model.predict(images, batch_size=10)
    print(fn)
    print(classes)
```

Note that it doesn't parse the output, just prints the classes. [Figure 3-20](#) shows what it looks like in use.

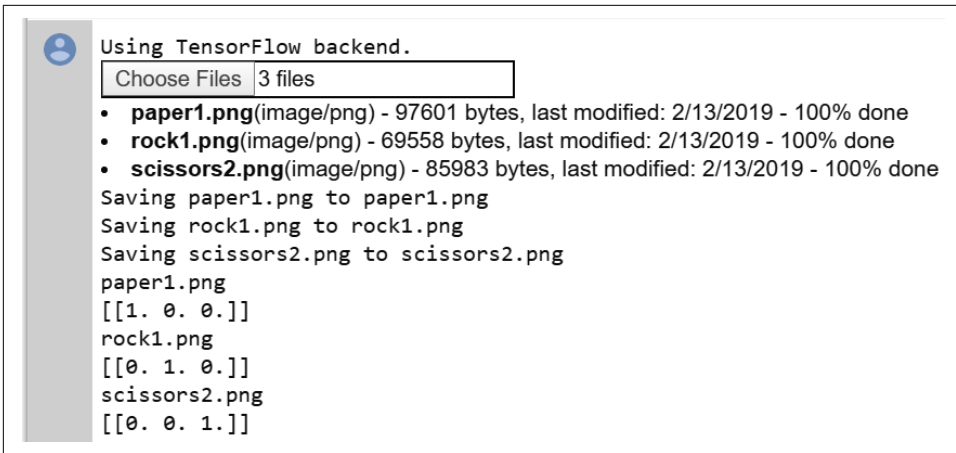


Figure 3-20. Testing the Rock/Paper/Scissors classifier

You can see from the filenames what the images were. *Paper1.png* ended up as `[1, 0, 0]`, meaning the first neuron was activated and the others weren't. Similarly, *Rock1.png* ended up as `[0, 1, 0]`, activating the second neuron, and *Scissors2.png* was `[0, 0, 1]`. Remember that the neurons are in alphabetical order by label!

Some images that you can use to test the dataset are available to [download](#). Alternatively, of course, you can try your own. Note that the training images are all done

against a plain white background, though, so there may be some confusion if there is a lot of detail in the background of the photos you take.

Dropout Regularization

Earlier in this chapter we discussed overfitting, where a network may become too specialized in a particular type of input data and fare poorly on others. One technique to help overcome this is use of *dropout regularization*.

When a neural network is being trained, each individual neuron will have an effect on neurons in subsequent layers. Over time, particularly in larger networks, some neurons can become overspecialized—and that feeds downstream, potentially causing the network as a whole to become overspecialized and leading to overfitting. Additionally, neighboring neurons can end up with similar weights and biases, and if not monitored this can lead the overall model to become overspecialized to the features activated by those neurons.

For example, consider the neural network in [Figure 3-21](#), where there are layers of 2, 6, 6, and 2 neurons. The neurons in the middle layers might end up with very similar weights and biases.

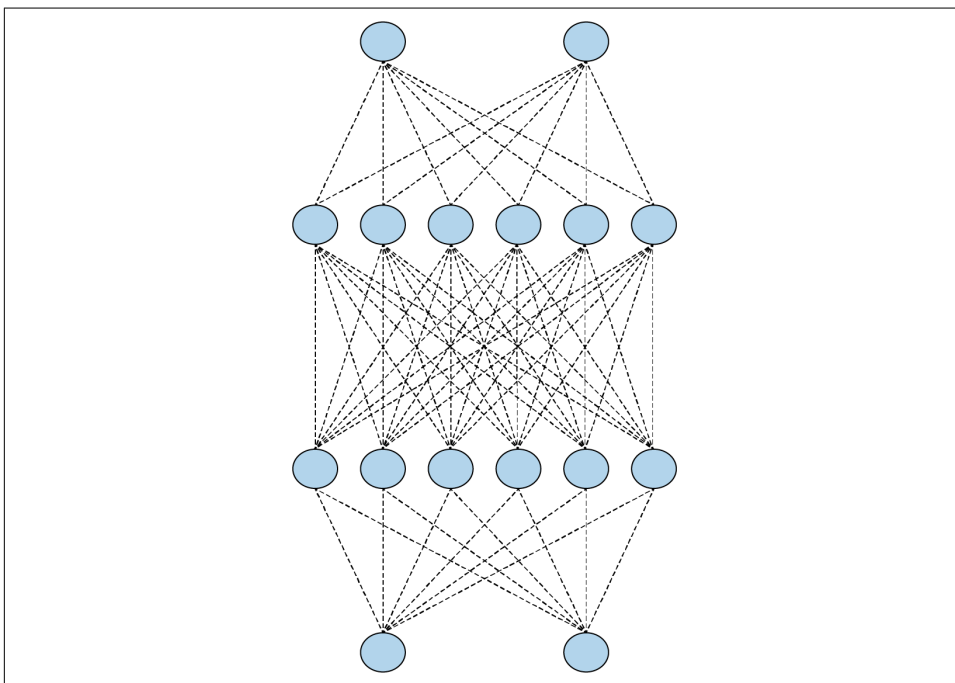


Figure 3-21. A simple neural network

While training, if you remove a random number of neurons and ignore them, their contribution to the neurons in the next layer is temporarily blocked (**Figure 3-22**).

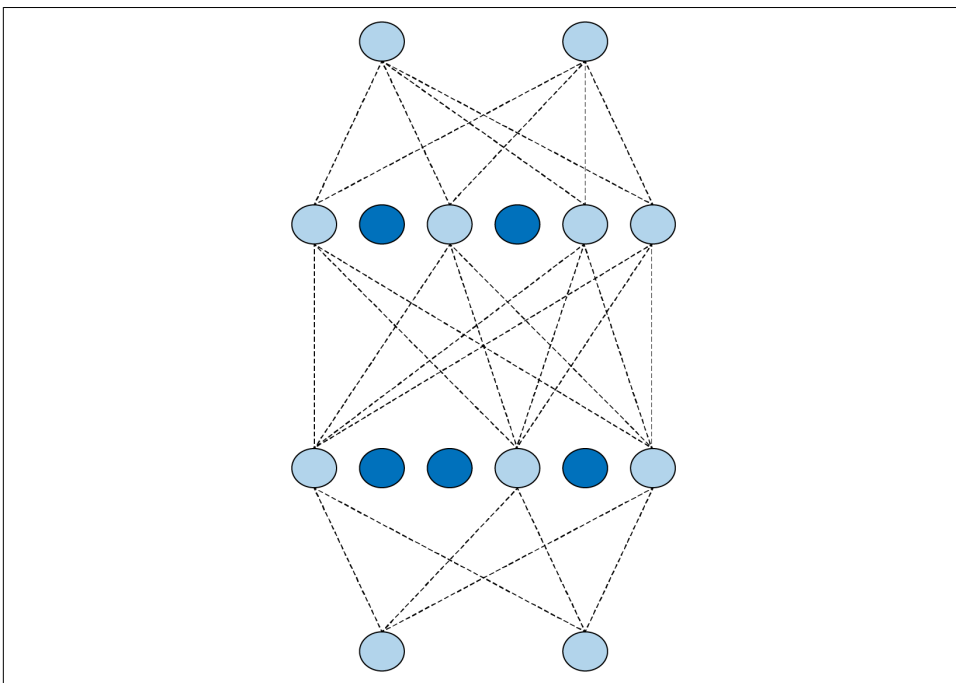


Figure 3-22. A neural network with dropouts

This reduces the chances of the neurons becoming overspecialized. The network will still learn the same number of parameters, but it should be better at generalization—that is, it should be more resilient to different inputs.



The concept of dropouts was proposed by Nitish Srivastava et al. in their 2014 paper “**Dropout: A Simple Way to Prevent Neural Networks from Overfitting**”.

To implement dropouts in TensorFlow, you can just use a simple Keras layer like this:

```
tf.keras.layers.Dropout(0.2),
```

This will drop out, at random, the specified percentage of neurons (here, 20%) in the specified layer. Note that it may take some experimentation to find the correct percentage for your network.

For a simple example that demonstrates this, consider the Fashion MNIST classifier from [Chapter 2](#). I'll change the network definition to have a lot more layers, like this:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28)),
    tf.keras.layers.Dense(256, activation=tf.nn.relu),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dense(64, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

Training this for 20 epochs gave around 94% accuracy on the training set, and about 88.5% on the validation set. This is a sign of potential overfitting.

Introducing dropouts after each dense layer looks like this:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28)),
    tf.keras.layers.Dense(256, activation=tf.nn.relu),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(64, activation=tf.nn.relu),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

When this network was trained for the same period on the same data, the accuracy on the training set dropped to about 89.5%. The accuracy on the validation set stayed about the same, at 88.3%. These values are much closer to each other; the introduction of dropouts thus not only demonstrated that overfitting was occurring, but also that using dropouts can help remove such ambiguity by ensuring that the network isn't overspecializing to the training data.

Keep in mind as you design your neural networks that great results on your training set are not always a good thing. This could be a sign of overfitting. Introducing dropouts can help you remove that problem, so that you can optimize your network in other areas without that false sense of security.

Summary

This chapter introduced you to a more advanced way of achieving computer vision using convolutional neural networks. You saw how to use convolutions to apply filters that can extract features from images, and designed your first neural networks to deal with more complex vision scenarios than those you encountered with the MNIST and Fashion MNIST datasets. You also explored techniques to improve your network's accuracy and avoid overfitting, such as the use of image augmentation and dropouts.

Before we explore further scenarios, in **Chapter 4** you'll get an introduction to TensorFlow Datasets, a technology that makes it much easier for you to get access to data for training and testing your networks. In this chapter you were downloading ZIP files and extracting images, but that's not always going to be possible. With TensorFlow Datasets you'll be able to access lots of datasets with a standard API.

Using Public Datasets with TensorFlow Datasets

In the first chapters of this book you trained models using a variety of data, from the Fashion MNIST dataset that is conveniently bundled with Keras to the image-based Horses or Humans and Dogs vs. Cats datasets, which were available as ZIP files that you had to download and preprocess. You’ve probably already realized that there are lots of different ways of getting the data with which to train a model.

However, many public datasets require you to learn lots of different domain-specific skills before you begin to consider your model architecture. The goal behind TensorFlow Datasets (TFDS) is to expose datasets in a way that’s easy to consume, where all the preprocessing steps of acquiring the data and getting it into TensorFlow-friendly APIs are done for you.

You’ve already seen a little of this idea with how Keras handled Fashion MNIST back in Chapters 1 and 2. As a recap, all you had to do to get the data was this:

```
data = tf.keras.datasets.fashion_mnist

(training_images, training_labels), (test_images, test_labels) =
data.load_data()
```

TFDS builds on this idea, but greatly expands not only the number of datasets available but the diversity of dataset types. The [list of available datasets](#) is growing all the time, in categories such as:

Audio

Speech and music data

Image

From simple learning datasets like Horses or Humans up to advanced research datasets for uses such as diabetic retinopathy detection

Object detection

COCO, Open Images, and more

Structured data

Titanic survivors, Amazon reviews, and more

Summarization

News from CNN and the *Daily Mail*, scientific papers, wikiHow, and more

Text

IMDb reviews, natural language questions, and more

Translate

Various translation training datasets

Video

Moving MNIST, Starcraft, and more



TensorFlow Datasets is a separate install from TensorFlow, so be sure to install it before trying out any samples! If you are using Google Colab, it's already preinstalled.

This chapter will introduce you to TFDS and how you can use it to greatly simplify the training process. We'll explore the underlying TFRecord structure and how it can provide commonality regardless of the type of the underlying data. You'll also learn about the Extract-Transform-Load (ETL) pattern using TFDS, which can be used to train models with huge amounts of data efficiently.

Getting Started with TFDS

Let's go through some simple examples of how to use TFDS to illustrate how it gives us a standard interface to our data, regardless of data type.

If you need to install it, you can do so with a `pip` command:

```
pip install tensorflow-datasets
```

Once it's installed, you can use it to get access to a dataset with `tlds.load`, passing it the name of the desired dataset. For example, if you want to use Fashion MNIST, you can use code like this:

```
import tensorflow as tf
import tensorflow_datasets as tfds
mnist_data = tfds.load("fashion_mnist")
for item in mnist_data:
    print(item)
```

Be sure to inspect the data type that you get in return from the `tfds.load` command—the output from printing the items will be the different splits that are natively available in the data. In this case it's a dictionary containing two strings, `test` and `train`. These are the available splits.

If you want to load these splits into a dataset containing the actual data, you can simply specify the split you want in the `tfds.load` command, like this:

```
mnist_train = tfds.load(name="fashion_mnist", split="train")
assert isinstance(mnist_train, tf.data.Dataset)
print(type(mnist_train))
```

In this instance, you'll see that the output is a `DatasetAdapter`, which you can iterate through to inspect the data. One nice feature of this adapter is you can simply call `take(1)` to get the first record. Let's do that to inspect what the data looks like:

```
for item in mnist_train.take(1):
    print(type(item))
    print(item.keys())
```

The output from the first `print` will show that the type of item in each record is a dictionary. When we print the keys to that we'll see that in this image set the types are `image` and `label`. So, if we want to inspect a value in the dataset, we can do something like this:

```
for item in mnist_train.take(1):
    print(type(item))
    print(item.keys())
    print(item['image'])
    print(item['label'])
```

You'll see the output for the image is a 28×28 array of values (in a `tf.Tensor`) from 0–255 representing the pixel intensity. The label will be output as `tf.Tensor(2, shape=(), dtype=int64)`, indicating that this image is class 2 in the dataset.

Data about the dataset is also available using the `with_info` parameter when loading the dataset, like this:

```
mnist_test, info = tfds.load(name="fashion_mnist", with_info="true")
print(info)
```

Printing the info will give you details about the contents of the dataset. For example, for Fashion MNIST, you'll see output like this:

```
tfds.core.DatasetInfo(
    name='fashion_mnist',
```

```

version=3.0.0,
description='Fashion-MNIST is a dataset of Zalando's article images
    consisting of a training set of 60,000 examples and a test set of 10,000
    examples. Each example is a 28x28 grayscale image, associated with a
    label from 10 classes.',
homepage='https://github.com/zalando-research/fashion-mnist',
features=FeaturesDict({
    'image': Image(shape=(28, 28, 1), dtype=tf.uint8),
    'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=10),
}),
total_num_examples=70000,
splits={
    'test': 10000,
    'train': 60000,
},
supervised_keys=('image', 'label'),
citation="""@article{DBLP:journals/corr/abs-1708-07747,
    author    = {Han Xiao and
                Kashif Rasul and
                Roland Vollgraf},
    title     = {Fashion-MNIST: a Novel Image Dataset for Benchmarking
                Machine Learning
                Algorithms},
    journal   = {CoRR},
    volume    = {abs/1708.07747},
    year      = {2017},
    url       = {http://arxiv.org/abs/1708.07747},
    archivePrefix = {arXiv},
    eprint    = {1708.07747},
    timestamp = {Mon, 13 Aug 2018 16:47:27 +0200},
    biburl    = {https://dblp.org/rec/bib/journals/corr/abs-1708-07747},
    bibsource = {dblp computer science bibliography, https://dblp.org}
}""",
redistribution_info=,
)

```

Within this you can see details such as the splits (as demonstrated earlier) and the features within the dataset, as well as extra information like the citation, description, and dataset version.

Using TFDS with Keras Models

In [Chapter 2](#) you saw how to create a simple computer vision model using TensorFlow and Keras, with the built-in datasets from Keras (including Fashion MNIST), using simple code like this:

```

mnist = tf.keras.datasets.fashion_mnist

(training_images, training_labels),
(test_images, test_labels) = mnist.load_data()

```

When using TFDS the code is very similar, but with some minor changes. The Keras datasets gave us ndarray types that worked natively in `model.fit`, but with TFDS we'll need to do a little conversion work:

```
(training_images, training_labels),
(test_images, test_labels) =
tfds.as_numpy(tfds.load('fashion_mnist',
                        split = ['train', 'test'],
                        batch_size=-1,
                        as_supervised=True))
```

In this case we use `tfds.load`, passing it `fashion_mnist` as the desired dataset. We know that it has train and test splits, so passing these in an array will return us an array of dataset adapters with the images and labels in them. Using `tfds.as_numpy` in the call to `tfds.load` causes them to be returned as Numpy arrays. Specifying `batch_size=-1` gives us *all* of the data, and `as_supervised=True` ensures we get tuples of (input, label) returned.

Once we've done that, we have pretty much the same format of data that was available in the Keras datasets, with one modification—the shape in TFDS is (28, 28, 1), whereas in the Keras datasets it was (28, 28).

This means the code needs to change a little to specify that the input data shape is (28, 28, 1) instead of (28, 28):

```
import tensorflow as tf
import tensorflow_datasets as tfds

(training_images, training_labels), (test_images, test_labels) =
tfds.as_numpy(tfds.load('fashion_mnist', split = ['train', 'test'],
batch_size=-1, as_supervised=True))

training_images = training_images / 255.0
test_images = test_images / 255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28,1)),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(training_images, training_labels, epochs=5)
```

For a more complex example, you can take a look at the Horses or Humans dataset used in [Chapter 3](#). This is also available in TFDS. Here's the complete code to train a model with it:

```

import tensorflow as tf
import tensorflow_datasets as tfds

data = tfds.load('horses_or_humans', split='train', as_supervised=True)

train_batches = data.shuffle(100).batch(10)

model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(16, (3,3), activation='relu',
                           input_shape=(300, 300, 3)),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

model.compile(optimizer='Adam', loss='binary_crossentropy',
              metrics=['accuracy'])

history = model.fit(train_batches, epochs=10)

```

As you can see, it's pretty straightforward: simply call `tfds.load`, passing it the split that you want (in this case `train`), and use that in the model. The data is batched and shuffled to make training more effective.

The Horses or Humans dataset is split into training and test sets, so if you want to do validation of your model while training, you can do so by loading a separate validation set from TFDS like this:

```
val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)
```

You'll need to batch it, the same as you did for the training set. For example:

```
validation_batches = val_data.batch(32)
```

Then, when training, you specify the validation data as these batches. You have to explicitly set the number of validation steps to use per epoch too, or TensorFlow will throw an error. If you're not sure, just set it to 1 like this:

```
history = model.fit(train_batches, epochs=10,
                    validation_data=validation_batches, validation_steps=1)
```

Loading Specific Versions

All datasets stored in TFDS use a *MAJOR.MINOR.PATCH* numbering system. The guarantees of this system are as follows. If *PATCH* is updated, then the data returned by a call is identical, but the underlying organization may have changed. Any changes should be invisible to developers. If *MINOR* is updated, then the data is still unchanged, with the exception that there may be additional features in each record (nonbreaking changes). Also, for any particular slice (see “Using Custom Splits” on page 74) the data will be the same, so records aren’t reordered. If *MAJOR* is updated, then there may be changes in the format of the records and their placement, so that particular slices may return different values.

When you inspect datasets, you will see when there are different versions available—for example, this is the case for the `cnn_dailymail` dataset. If you don’t want the default one, which at time of writing was 3.0.0, and instead want an earlier one, such as 1.0.0, you can simply load it like this:

```
data, info = tfds.load("cnn_dailymail:1.0.0", with_info=True)
```

Note that if you are using Colab, it’s always a good idea to check the version of TFDS that it uses. At time of writing, Colab was preconfigured for TFDS 2.0, but there are some bugs in loading datasets (including the `cnn_dailymail` one) that have been fixed in TFDS 2.1 and later, so be sure to use one of those versions, or at least install them into Colab, instead of relying on the built-in default.

Using Mapping Functions for Augmentation

In [Chapter 3](#) you saw the useful augmentation tools that were available when using an `ImageDataGenerator` to provide the training data for your model. You may be wondering how you might achieve the same when using TFDS, as you aren’t flowing the images from a subdirectory like before. The best way to achieve this—or indeed any other form of transformation—is to use a mapping function on the data adapter. Let’s take a look at how to do that.

Earlier, with our Horses or Humans data, we simply loaded the data from TFDS and created batches for it like this:

```
data = tfds.load('horses_or_humans', split='train', as_supervised=True)

train_batches = data.shuffle(100).batch(10)
```

To do transforms and have them mapped to the dataset, you can create a *mapping function*. This is just standard Python code. For example, suppose you create a function called `augmentimages` and have it do some image augmentation, like this:

```
def augmentimages(image, label):
    image = tf.cast(image, tf.float32)
```



```
image = (image/255)
image = tf.image.random_flip_left_right(image)
return image, label
```

You can then map this to the data to create a new dataset called `train`:

```
train = data.map(augmentimages)
```

Then when you create the batches, do this from `train` instead of from `data`, like this:

```
train_batches = train.shuffle(100).batch(32)
```

You can see in the `augmentimages` function that there is a random flip left or right of the image, done using `tf.image.random_flip_left_right(image)`. There are lots of functions in the `tf.image` library that you can use for augmentation; see the [documentation](#) for details.

Using TensorFlow Addons

The [TensorFlow Addons](#) library contains even more functions that you can use. Some of the functions in the `ImageDataGenerator` augmentation (such as `rotate`) can only be found there, so it's a good idea to check it out.

Using TensorFlow Addons is pretty easy—you simply install the library with:

```
pip install tensorflow-addons
```

Once that's done, you can mix the addons into your mapping function. Here's an example where the `rotate` addon is used in the mapping function from earlier:

```
import tensorflow_addons as tfa

def augmentimages(image, label):
    image = tf.cast(image, tf.float32)
    image = (image/255)
    image = tf.image.random_flip_left_right(image)
    image = tfa.image.rotate(image, 40, interpolation='NEAREST')
    return image, label
```

Using Custom Splits

Up to this point, all of the data you've been using to build models has been presplit into training and test sets for you. For example, with Fashion MNIST you had 60,000 and 10,000 records, respectively. But what if you don't want to use those splits? What if you want to split the data yourself according to your own needs? That's one of the aspects of TFDS that's really powerful—it comes complete with an API that gives you fine, granular control over how you split your data.

You've actually seen it already when loading data like this:

```
data = tfds.load('cats_vs_dogs', split='train', as_supervised=True)
```

Note that the `split` parameter is a string, and in this case you're asking for the `train` split, which happens to be the entire dataset. If you're familiar with **Python slice notation**, you can use that as well. This notation can be summarized as defining your desired slices within square brackets like this: `[<start>: <stop>: <step>]`. It's quite a sophisticated syntax giving you great flexibility.

For example, if you want the first 10,000 records of `train` to be your training data, you can omit `<start>` and just call for `train[:10000]` (a useful mnemonic is to read the leading colon as “the first,” so this would read “train the first 10,000 records”):

```
data = tfds.load('cats_vs_dogs', split='train[:10000]', as_supervised=True)
```

You can also use `%` to specify the split. For example, if you want the first 20% of the records to be used for training, you could use `:20%` like this:

```
data = tfds.load('cats_vs_dogs', split='train[:20%]', as_supervised=True)
```

You could even get a little crazy and combine splits. That is, if you want your training data to be a combination of the first and last thousand records, you could do the following (where `-1000:` means “the last 1,000 records” and `:1000` means “the first 1,000 records”):

```
data = tfds.load('cats_vs_dogs', split='train[-1000:]+train[:1000]',
                 as_supervised=True)
```

The Dogs vs. Cats dataset doesn't have fixed training, test, and validation splits, but, with TFDS, creating your own is simple. Suppose you want the split to be 80%, 10%, 10%. You could create the three sets like this:

```
train_data = tfds.load('cats_vs_dogs', split='train[:80%]',
                      as_supervised=True)
```

```
validation_data = tfds.load('cats_vs_dogs', split='train[80%:90%]',
                           as_supervised=True)
```

```
test_data = tfds.load('cats_vs_dogs', split='train[-10%:]',
                    as_supervised=True)
```

Once you have them, you can use them as you would any named split.

One caveat is that because the datasets that are returned can't be interrogated for length, it's often difficult to check that you have split the original set correctly. To see how many records you have in a split, you have to iterate through the whole set and count them one by one. Here's the code to do that for the training set you just created:

```
train_length = [i for i, _ in enumerate(train_data)][-1] + 1
print(train_length)
```

This can be a slow process, so be sure to use it only when you're debugging!

Understanding TFRecord

When you're using TFDS, your data is downloaded and cached to disk so that you don't need to download it each time you use it. TFDS uses the TFRecord format for caching. If you watch closely as it's downloading the data you'll see this—for example, [Figure 4-1](#) shows how the `cnn_dailymail` dataset is downloaded, shuffled, and written to a TFRecord file.

```
/tensorflow_datasets/cnn_dailymail/plain_text/1.0.0.incompleteFFCKFG/cnn_dailymail-train.tfrecord
99% 282918/287113 [00:06<00:00, 48503.77 examples/s]

10153 examples [00:07, 1475.51 examples/s]
```

Figure 4-1. Downloading the `cnn_dailymail` dataset as a TFRecord file

This is the preferred format in TensorFlow for storing and retrieving large amounts of data. It's a very simple file structure, read sequentially for better performance. On disk the file is pretty straightforward, with each record consisting of an integer indicating the length of the record, a cyclic redundancy check (CRC) of that, a byte array of the data, and a CRC of that byte array. The records are concatenated into the file and then sharded in the case of large datasets.

For example, [Figure 4-2](#) shows how the training set from `cnn_dailymail` is sharded into 16 files after download.

To take a look at a simpler example, download the MNIST dataset and print its info:

```
data, info = tfds.load("mnist", with_info=True)
print(info)
```

Within the info you'll see that its features are stored like this:

```
features=FeaturesDict({
  'image': Image(shape=(28, 28, 1), dtype=tf.uint8),
  'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=10),
})
```

Similar to the CNN/DailyMail example, the file is downloaded to `/root/tensorflow_datasets/mnist/<version>/files`.

You can load the raw records as a `TFRecordDataset` like this:

```
filename="/root/tensorflow_datasets/mnist/3.0.0/
mnist-test.tfrecord-00000-of-00001"
raw_dataset = tf.data.TFRecordDataset(filename)
for raw_record in raw_dataset.take(1):

    print(repr(raw_record))
```

Note that your filename location may be different depending on your operating system.

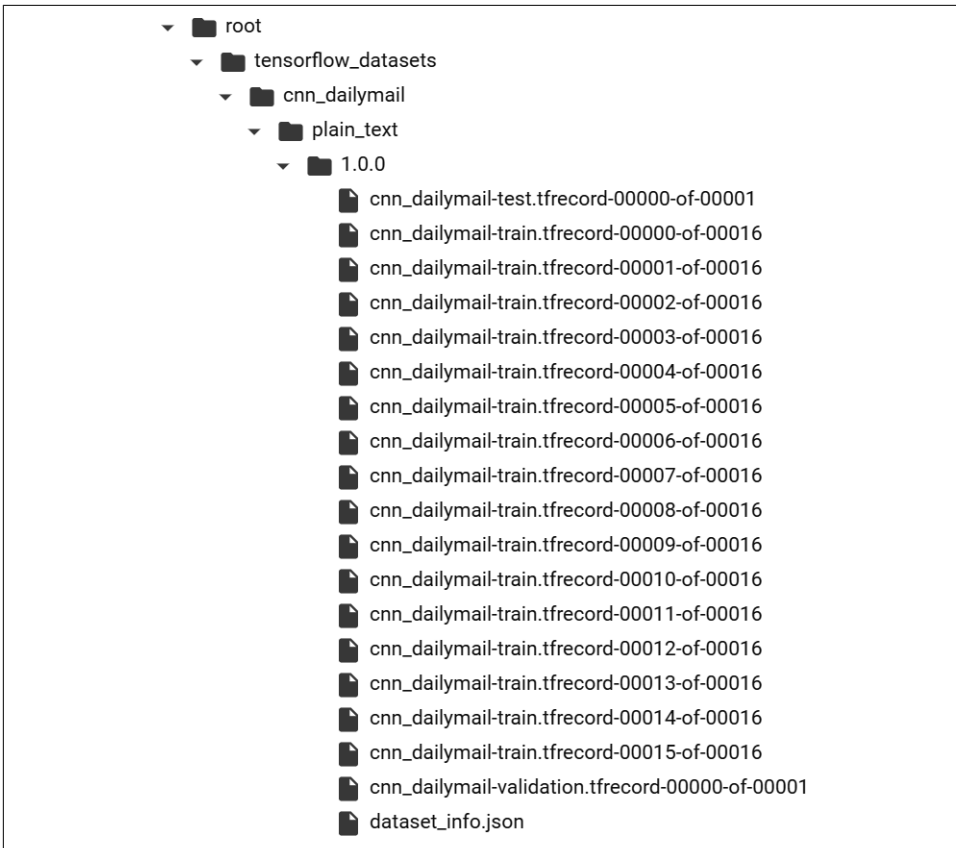


Figure 4-2. Inspecting the TFRecords for *cnn_dailymail*

This will print out the raw contents of the record, like this:

```

<tf.Tensor: shape=(), dtype=string,
numpy=b"\n\x85\x03\n\xf2\x02\n\x05image\x12\xe8\x02\n\xe5\x02\n\xe2\x02\x89PNG\r
\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\x00\x1c\x00\x00\x00\x1c\x08\x00\x00\x00Wf
\x80H\x00\x00\x01)IDAT(\x91\xc5\xd2\xbdK\xc3P\x14\x05\xf05(v\x13)\x04,.\x82\xc5A
q\xac\xedb\x1d\xdc\n.\x12\x87n\x0e\x82\x93\x7f@Q\xb2\x08\xba\tbQ0.\xe2\xe2\xd4\x
b1\xa2h\x9c\x82\xba\x8a(\nq\xf0\x83Fh\x95\n6\x88\xe7R\x87\x88\xf9\xa8Y\xf5\x0e\x
8f\xc7\xfd\xdd\x0b\x87\xc7\x03\xfe\xbeb\x9d\xadT\x927Q\xe3\xe9\x07:\xab\xbf\xf4\
xf3\xcf\xf6\x8a\xd9\x14\xd29\xea\xb0\x1eKH\xde\xab\xea%\xab\x1b=\xa4P/\xf5\x02\
xd7||\x07\x00\xc4=,L\x00,>\x01@2\xf6\x12\xde\x9c\xde[t/\xb3\x0e\x87\xa2\xe2\
xc2\xe0A<\xca\xb26\xd5(\x1b\xa9\xd3\xe8\x0e\xf5\x86\x17\xceE\xdarV\xae\xb7_\xf3
I\xf7(\x06m\xaaE\xbb\xbb\xac\r*\x9b$e<\xb8\xd7\xa2\x0e\x00\xd0l\x92\xb2\xd5\x15\
xcc\xae'\x00\xf4m\x080'+\xc2y\x9f\x8d\xc9\x15\x80\xfe\x99[q\x962@CN|i\xf7\xa9!=\
\xab\x19\x00\xc8\xd6\xb8\xeb\xa1\xf0\xd8l\xca\xfb]\xee\xfb]*\x9fV\x01\x07\xb7\x
9\x8b55\xe7M\xef\xb0\x04\x00\xfd8\x89\x01<\xbe\xf9\x03*\x8a\xf5\x81\x7f\xaa/2y\x
87ks\xec\x1e\x01\x00\x00\x00\x00\x00IEND\xaeB`\x82\n\x0e\n\x05label\x12\x05\x1a\x03\
n\x01\x02">

```

It's a long string containing the details of the record, along with checksums, etc. But if we already know the features, we can create a feature description and use this to parse the data. Here's the code:

```
# Create a description of the features
feature_description = {
    'image': tf.io.FixedLenFeature([], dtype=tf.string),
    'label': tf.io.FixedLenFeature([], dtype=tf.int64),
}

def _parse_function(example_proto):
    # Parse the input `tf.Example` proto using the dictionary above
    return tf.io.parse_single_example(example_proto, feature_description)

parsed_dataset = raw_dataset.map(_parse_function)
for parsed_record in parsed_dataset.take(1):
    print((parsed_record))
```

The output of this is a little friendlier! First of all, you can see that the image is a Tensor, and that it contains a PNG. PNG is a compressed image format with a header defined by IHDR and the image data between IDAT and IEND. If you look closely, you can see them in the byte stream. There's also the label, stored as an int and containing the value 2:

```
{'image': <tf.Tensor: shape=(), dtype=string,
numpy=b"\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\x1c\x00\x00\x00\x1c\x08\x00\x00\x00\x00Wf\x80H\x00\x00\x01)IDAT(\x91\xc5\xd2\xbdK\xc3P\x14\x05\xf0S(v\x13)\x04,.\x82\xc5Aq\xac\xedb\x1d\xdc\n.\x12\x87n\x0e\x82\x93\x7f@Q\xb2\x08\xba\tbQ0.\xe2\xe2\xd4\xb1\xa2h\x9c\x82\xba\x8a(\nq\xfc\x83Fh\x95\n6\x88\xe7R\x87\x88\xf9\xa8Y\xf5\x0e\x8f\xc7\xfd\xdd\x0b\x87\xc7\x03\xfe\xbeb\x9d\xadT\x927Q\xe3\xe9\x07:\xab\xbf\xf4\xf3\xcf\xf6\x8a\xd9\x14\xd29\xea\xb0\x1eKH\xde\xab\xea%\xab\x1b=\xa4P/\xf5\x02\xd7|\x07\x00\xc4=,L\x00,>\x01@2\xf6\x12\xde\x9c\xde[t/\xb3\x0e\x87\xa2\xe2\xc2\xe0A<\xca\xb26\xd5(\x1b\xa9\xd3\xe8\x0e\xf5\x86\x17\xceE\xdarV\xae\xbb7_\xf3AR\r!I\xf7(\x06m\xaaE\xbb\xbb\xac\r*\x9b$e<\xb8\xd7\xa2\x0e\x00\xd0l\x92\xb2\xd5\x15\xcc\xae'\x00\xf4m\x080'+\xc2y\x9f\x8d\xc9\x15\x80\xfe\x99[q\x962@CN|i\xfc7\xa9!:=\xd7\xab\x19\x00\xc8\xd6\xb8\xeb\xa1\xfc\xd8l\xca\xfb]\xee\xfb]*\x9fV\xe1\x07\xb7\x99\x8b55\xe7M\xef\xb0\x04\x00\xfd8\x89\x01<\xbe\xfc9\x03*\x8a\xfc5\x81\x7f\xaa/2y\x87ks\xec\x1e\x00\x00\x00\x00IEND\xaeB`\x82">, 'label': <tf.Tensor: shape=(), dtype=int64, numpy=2>}
```

At this point you can read the raw TFRecord and decode it as a PNG using a PNG decoder library like Pillow.

The ETL Process for Managing Data in TensorFlow

ETL is the core pattern that TensorFlow uses for training, regardless of scale. We've been exploring small-scale, single-computer model building in this book, but the same technology can be used for large-scale training across multiple machines with massive datasets.

The *Extract* phase of the ETL process is when the raw data is loaded from wherever it is stored and prepared in a way that can be transformed. The *Transform* phase is when the data is manipulated in a way that makes it suitable or improved for training. For example, batching, image augmentation, mapping to feature columns, and other such logic applied to the data can be considered part of this phase. The *Load* phase is when the data is loaded into the neural network for training.

Consider the full code to train the Horses or Humans classifier, shown here. I've added comments to show where the Extract, Transform, and Load phases take place:

```
import tensorflow as tf
import tensorflow_datasets as tfds
import tensorflow_addons as tfa

# MODEL DEFINITION START #
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(16, (3,3), activation='relu',
                           input_shape=(300, 300, 3)),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(optimizer='Adam', loss='binary_crossentropy',
              metrics=['accuracy'])
# MODEL DEFINITION END #

# EXTRACT PHASE START #
data = tfds.load('horses_or_humans', split='train', as_supervised=True)
val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)
# EXTRACT PHASE END

# TRANSFORM PHASE START #
def augmentimages(image, label):
    image = tf.cast(image, tf.float32)
    image = (image/255)
    image = tf.image.random_flip_left_right(image)
    image = tfa.image.rotate(image, 40, interpolation='NEAREST')
    return image, label

train = data.map(augmentimages)
train_batches = train.shuffle(100).batch(32)
validation_batches = val_data.batch(32)
```

```
# TRANSFORM PHASE END

# LOAD PHASE START #
history = model.fit(train_batches, epochs=10,
                    validation_data=validation_batches, validation_steps=1)
# LOAD PHASE END #
```

Using this process can make your data pipelines less susceptible to changes in the data and the underlying schema. When you use TFDS to extract data, the same underlying structure is used regardless of whether the data is small enough to fit in memory, or so large that it cannot be contained even on a simple machine. The `tf.data` APIs for transformation are also consistent, so you can use similar ones regardless of the underlying data source. And, of course, once it's transformed, the process of loading the data is also consistent whether you are training on a single CPU, a GPU, a cluster of GPUs, or even pods of TPUs.

How you load the data, however, can have a huge impact on your training speed. Let's take a look at that next.

Optimizing the Load Phase

Let's take a closer look at the Extract-Transform-Load process when training a model. We can consider the extraction and transformation of the data to be possible on any processor, including a CPU. In fact, the code used in these phases to perform tasks like downloading data, unzipping it, and going through it record by record and processing them is not what GPUs or TPUs are built for, so this code will likely execute on the CPU anyway. When it comes to training, however, you can get great benefits from a GPU or TPU, so it makes sense to use one for this phase if possible. Thus, in the situation where a GPU or TPU is available to you, you should ideally split the workload between the CPU and the GPU/TPU, with Extract and Transform taking place on the CPU, and Load taking place on the GPU/TPU.

Suppose you're working with a large dataset. Assuming it's so large that you have to prepare the data (i.e., do the extraction and transformation) in batches, you'll end up with a situation like that shown in [Figure 4-3](#). While the first batch is being prepared, the GPU/TPU is idle. When that batch is ready it can be sent to the GPU/TPU for training, but now the CPU is idle until the training is done, when it can start preparing the second batch. There's a lot of idle time here, so we can see that there's room for optimization.

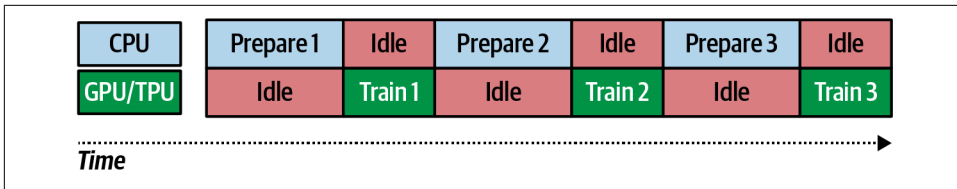


Figure 4-3. Training on a CPU/GPU

The logical solution is to do the work in parallel, preparing and training side by side. This process is called *pipelining* and is illustrated in Figure 4-4.

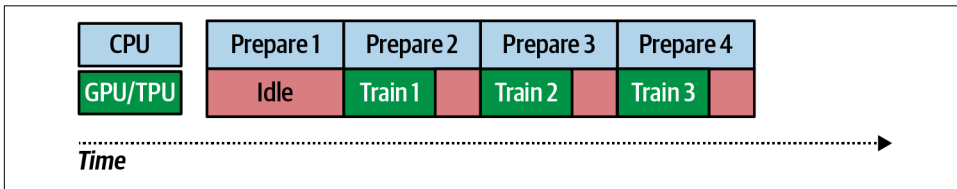


Figure 4-4. Pipelining

In this case, while the CPU prepares the first batch the GPU/TPU again has nothing to work on, so it's idle. When the first batch is done, the GPU/TPU can start training—but in parallel with this, the CPU will prepare the second batch. Of course, the time it takes to train batch $n - 1$ and prepare batch n won't always be the same. If the training time is faster, you'll have periods of idle time on the GPU/TPU. If it's slower, you'll have periods of idle time on the CPU. Choosing the correct batch size can help you optimize here—and as GPU/TPU time is likely more expensive, you'll probably want to reduce its idle time as much as possible.

You probably noticed when we moved from simple datasets like Fashion MNIST in Keras to using the TFDS versions that you had to batch them before you could train. This is why: the pipelining model is in place so that regardless of how large your dataset is, you'll continue to use a consistent pattern for ETL on it.

Parallelizing ETL to Improve Training Performance

TensorFlow gives you all the APIs you need to parallelize the Extract and Transform process. Let's explore what they look like using Dogs vs. Cats and the underlying TFRecord structures.

First, you use `tfds.load` to get the dataset:

```
train_data = tfds.load('cats_vs_dogs', split='train', with_info=True)
```

If you want to use the underlying TFRecords, you'll need to access the raw files that were downloaded. As the dataset is large, it's sharded across a number of files (eight, in version 4.0.0).

You can create a list of these files and use `tf.Data.Dataset.list_files` to load them:

```
file_pattern =
    f'/root/tensorflow_datasets/cats_vs_dogs/4.0.0/cats_vs_dogs-train.tfrecord*'
files = tf.data.Dataset.list_files(file_pattern)
```

Once you have the files, they can be loaded into a dataset using `files.interleave` like this:

```
train_dataset = files.interleave(
    tf.data.TFRecordDataset,
    cycle_length=4,
    num_parallel_calls=tf.data.experimental.AUTOTUNE
)
```

There are a few new concepts here, so let's take a moment to explore them.

The `cycle_length` parameter specifies the number of input elements that are processed concurrently. So, in a moment you'll see the mapping function that decodes the records as they're loaded from disk. Because `cycle_length` is set to 4, this process will be handling four records at a time. If you don't specify this value, then it will be derived from the number of available CPU cores.

The `num_parallel_calls` parameter, when set, will specify the number of parallel calls to execute. Using `tf.data.experimental.AUTOTUNE`, as is done here, will make your code more portable because the value is set dynamically, based on the available CPUs. When combined with `cycle_length`, you're setting the maximum degree of parallelism. So, for example, if `num_parallel_calls` is set to 6 after autotuning and `cycle_length` is 4, you'll have six separate threads, each loading four records at a time.

Now that the Extract process is parallelized, let's explore parallelizing the transformation of the data. First, create the mapping function that loads the raw `TFRecord` and converts it to usable content—for example, decoding a JPEG image into an image buffer:

```
def read_tfrecord(serialized_example):
    feature_description={
        "image": tf.io.FixedLenFeature((), tf.string, ""),
        "label": tf.io.FixedLenFeature((), tf.int64, -1),
    }
    example = tf.io.parse_single_example(
        serialized_example, feature_description
    )
    image = tf.io.decode_jpeg(example['image'], channels=3)
    image = tf.cast(image, tf.float32)
    image = image / 255
    image = tf.image.resize(image, (300,300))
    return image, example['label']
```

As you can see, this is a typical mapping function without any specific work done to make it work in parallel. That will be done when we call the mapping function. Here's how to do that:

```
cores = multiprocessing.cpu_count()
print(cores)
train_dataset = train_dataset.map(read_tfrecord, num_parallel_calls=cores)
train_dataset = train_dataset.cache()
```

First, if you don't want to autotune, you can use the `multiprocessing` library to get a count of your CPUs. Then, when you call the mapping function, you just pass this as the number of parallel calls that you want to make. It's really as simple as that.

The `cache` method will cache the dataset in memory. If you have a lot of RAM available this is a really useful speedup. Trying this in Colab with Dogs vs. Cats will likely crash your VM due to the dataset not fitting in RAM. After that, if available, the Colab infrastructure will give you a new, higher-RAM machine.

Loading and training can also be parallelized. As well as shuffling and batching the data, you can prefetch based on the number of CPU cores that are available. Here's the code:

```
train_dataset = train_dataset.shuffle(1024).batch(32)
train_dataset = train_dataset.prefetch(tf.data.experimental.AUTOTUNE)
```

Once your training set is all parallelized, you can train the model as before:

```
model.fit(train_dataset, epochs=10, verbose=1)
```

When I tried this in Google Colab, I found that this extra code to parallelize the ETL process reduced the training time to about 40 seconds per epoch, as opposed to 75 seconds without it. These simple changes cut my training time almost in half!

Summary

This chapter introduced TensorFlow Datasets, a library that gives you access to a huge range of datasets, from small learning ones up to full-scale datasets used in research. You saw how they use a common API and common format to help reduce the amount of code you have to write to get access to data. You also saw how to use the ETL process, which is at the heart of the design of TFDS, and in particular we explored parallelizing the extraction, transformation, and loading of data to improve training performance. In the next chapter you'll take what you've learned and start applying it to natural language processing problems.

Introduction to Natural Language Processing

Natural language processing (NLP) is a technique in artificial intelligence that deals with the understanding of human-based language. It involves programming techniques to create a model that can understand language, classify content, and even generate and create new compositions in human-based language. We'll be exploring these techniques over the next few chapters. There are also lots of services that use NLP to create applications such as chatbots, but that's not in the scope of this book—instead, we'll be looking at the foundations of NLP and how to model language so that you can train neural networks to understand and classify text. For a little fun, you'll also see how to use the predictive elements of a machine learning model to write some poetry!

We'll start this chapter by looking at how to decompose language into numbers, and how those numbers can then be used in neural networks.

Encoding Language into Numbers

You can encode language into numbers in many ways. The most common is to encode by letters, as is done naturally when strings are stored in your program. In memory, however, you don't store the letter *a* but an encoding of it—perhaps an ASCII or Unicode value, or something else. For example, consider the word *listen*. This can be encoded with ASCII into the numbers 76, 73, 83, 84, 69, and 78. This is good, in that you can now use numerics to represent the word. But then consider the word *silent*, which is an antigram of *listen*. The same numbers represent that word, albeit in a different order, which might make building a model to understand the text a little difficult.



An *antigram* is a word that's an anagram of another, but has the opposite meaning. For example, *united* and *untied* are antigrams, as are *restful* and *fluster*, *Santa* and *Satan*, *forty-five* and *over fifty*. My job title used to be Developer Evangelist but has since changed to Developer Advocate—which is a good thing because *Evangelist* is an antigram for *Evil's Agent*!

A better alternative might be to use numbers to encode entire words instead of the letters within them. In that case, *silent* could be number x and *listen* number y , and they wouldn't overlap with each other.

Using this technique, consider a sentence like “I love my dog.” You could encode that with the numbers [1, 2, 3, 4]. If you then wanted to encode “I love my cat.” it could be [1, 2, 3, 5]. You've already gotten to the point where you can tell that the sentences have a similar meaning because they're similar numerically—[1, 2, 3, 4] looks a lot like [1, 2, 3, 5].

This process is called *tokenization*, and you'll explore how to do that in code next.

Getting Started with Tokenization

TensorFlow Keras contains a library called preprocessing that provides a number of extremely useful tools to prepare data for machine learning. One of these is a *Tokenizer* that will allow you to take words and turn them into tokens. Let's see it in action with a simple example:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.text import Tokenizer

sentences = [
    'Today is a sunny day',
    'Today is a rainy day'
]

tokenizer = Tokenizer(num_words = 100)
tokenizer.fit_on_texts(sentences)
word_index = tokenizer.word_index
print(word_index)
```

In this case, we create a *Tokenizer* object and specify the number of words that it can tokenize. This will be the maximum number of tokens to generate from the corpus of words. We have a very small corpus here containing only six unique words, so we'll be well under the one hundred specified.

Once we have a tokenizer, calling `fit_on_texts` will create the tokenized word index. Printing this out will show a set of key/value pairs for the words in the corpus, like this:

```
{'today': 1, 'is': 2, 'a': 3, 'day': 4, 'sunny': 5, 'rainy': 6}
```

The tokenizer is quite flexible. For example, if we were to expand the corpus with another sentence containing the word “today” but with a question mark after it, the results show that it would be smart enough to filter out “today?” as just “today”:

```
sentences = [  
    'Today is a sunny day',  
    'Today is a rainy day',  
    'Is it sunny today?'  
]  
  
{'today': 1, 'is': 2, 'a': 3, 'sunny': 4, 'day': 5, 'rainy': 6, 'it': 7}
```

This behavior is controlled by the `filters` parameter to the tokenizer, which defaults to removing all punctuation except the apostrophe character. So for example, “Today is a sunny day” would become a sequence containing [1, 2, 3, 4, 5] with the preceding encodings, and “Is it sunny today?” would become [2, 7, 4, 1]. Once you have the words in your sentences tokenized, the next step is to convert your sentences into lists of numbers, with the number being the value where the word is the key.

Turning Sentences into Sequences

Now that you’ve seen how to take words and tokenize them into numbers, the next step is to encode the sentences into sequences of numbers. The tokenizer has a method for this called `text_to_sequences`—all you have to do is pass it your list of sentences, and it will give you back a list of sequences. So, for example, if you modify the preceding code like this:

```
sentences = [  
    'Today is a sunny day',  
    'Today is a rainy day',  
    'Is it sunny today?'  
]  
  
tokenizer = Tokenizer(num_words = 100)  
tokenizer.fit_on_texts(sentences)  
word_index = tokenizer.word_index  
  
sequences = tokenizer.texts_to_sequences(sentences)  
  
print(sequences)
```

you’ll be given the sequences representing the three sentences. Remembering that the word index is this:

```
{'today': 1, 'is': 2, 'a': 3, 'sunny': 4, 'day': 5, 'rainy': 6, 'it': 7}
```

the output will look like this:

```
[[1, 2, 3, 4, 5], [1, 2, 3, 6, 5], [2, 7, 4, 1]]
```

You can then substitute in the words for the numbers and you'll see that the sentences make sense.

Now consider what happens if you are training a neural network on a set of data. The typical pattern is that you have a set of data used for training that you know won't cover 100% of your needs, but you hope covers as much as possible. In the case of NLP, you might have many thousands of words in your training data, used in many different contexts, but you can't have every possible word in every possible context. So when you show your neural network some new, previously unseen text, containing previously unseen words, what might happen? You guessed it—it will get confused because it simply has no context for those words, and, as a result, any prediction it gives will be negatively affected.

Using out-of-vocabulary tokens

One tool to use to handle these situations is an *out-of-vocabulary* (OOV) token. This can help your neural network to understand the context of the data containing previously unseen text. For example, given the previous small example corpus, suppose you want to process sentences like these:

```
test_data = [  
    'Today is a snowy day',  
    'Will it be rainy tomorrow?'  
]
```

Remember that you're not adding this input to the corpus of existing text (which you can think of as your training data), but considering how a pretrained network might view this text. If you tokenize it with the words that you've already used and your existing tokenizer, like this:

```
test_sequences = tokenizer.texts_to_sequences(test_data)  
print(word_index)  
print(test_sequences)
```

Your results will look like this:

```
{'today': 1, 'is': 2, 'a': 3, 'sunny': 4, 'day': 5, 'rainy': 6, 'it': 7}  
[[1, 2, 3, 5], [7, 6]]
```

So the new sentences, swapping back tokens for words, would be “today is a day” and “it rainy.”

As you can see, you've pretty much lost all context and meaning. An out-of-vocabulary token might help here, and you can specify it in the tokenizer. You do this by adding a parameter called `oov_token`, as shown here—you can assign it any string you like, but make sure it's not one that appears elsewhere in your corpus:

```
tokenizer = Tokenizer(num_words = 100, oov_token="<OOV>")  
tokenizer.fit_on_texts(sentences)  
word_index = tokenizer.word_index
```

```

sequences = tokenizer.texts_to_sequences(sentences)

test_sequences = tokenizer.texts_to_sequences(test_data)
print(word_index)
print(test_sequences)

```

You'll see the output has improved a bit:

```

{'<OOV>': 1, 'today': 2, 'is': 3, 'a': 4, 'sunny': 5, 'day': 6, 'rainy': 7,
 'it': 8}

[[2, 3, 4, 1, 6], [1, 8, 1, 7, 1]]

```

Your tokens list has a new item, “<OOV>,” and your test sentences maintain their length. Reverse-encoding them will now give “today is a <OOV> day” and “<OOV> it <OOV> rainy <OOV>.”

The former is much closer to the original meaning. The latter, because most of its words aren't in the corpus, still lacks a lot of context, but it's a step in the right direction.

Understanding padding

When training neural networks you typically need all your data to be in the same shape. Recall from earlier chapters that when training with images, you reformatted the images to be the same width and height. With text you face the same issue—once you've tokenized your words and converted your sentences into sequences, they can all be different lengths. To get them to be the same size and shape, you can use *padding*.

To explore padding, let's add another, much longer, sentence to the corpus:

```

sentences = [
    'Today is a sunny day',
    'Today is a rainy day',
    'Is it sunny today?',
    'I really enjoyed walking in the snow today'
]

```

When you sequence that, you'll see that your lists of numbers have different lengths:

```

[
    [2, 3, 4, 5, 6],
    [2, 3, 4, 7, 6],
    [3, 8, 5, 2],
    [9, 10, 11, 12, 13, 14, 15, 2]
]

```

(When you print the sequences they'll all be on a single line, but I've broken them into separate lines here for clarity.)

If you want to make these the same length, you can use the `pad_sequences` API. First, you'll need to import it:

```
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

Using the API is very straightforward. To convert your (unpadded) sequences into a padded set, you simply call `pad_sequences` like this:

```
padded = pad_sequences(sequences)
```

```
print(padded)
```

You'll get a nicely formatted set of sequences. They'll also be on separate lines, like this:

```
[[ 0  0  0  2  3  4  5  6]
 [ 0  0  0  2  3  4  7  6]
 [ 0  0  0  0  3  8  5  2]
 [ 9 10 11 12 13 14 15  2]]
```

The sequences get padded with 0, which isn't a token in our word list. If you had wondered why the token list began at 1 when typically programmers count from 0, now you know!

You now have something that's regularly shaped that you can use for training. But before going there, let's explore this API a little, because it gives you many options that you can use to improve your data.

First, you might have noticed that in the case of the shorter sentences, to get them to be the same shape as the longest one, the requisite number of zeros were added at the beginning. This is called *prepadding*, and it's the default behavior. You can change this using the `padding` parameter. For example, if you want your sequences to be padded with zeros at the end, you can use:

```
padded = pad_sequences(sequences, padding='post')
```

The output from this will be:

```
[[ 2  3  4  5  6  0  0  0]
 [ 2  3  4  7  6  0  0  0]
 [ 3  8  5  2  0  0  0  0]
 [ 9 10 11 12 13 14 15  2]]
```

You can see that now the words are at the beginning of the padded sequences, and the 0 characters are at the end.

The next default behavior you may have observed is that the sentences were all made to be the same length as the *longest* one. It's a sensible default because it means you don't lose any data. The trade-off is you get a lot of padding. But what if you don't want this, perhaps because you have one crazy long sentence that means you'd have too much padding in the padded sequences? To fix that you can use the `maxlen`

parameter, specifying the desired maximum length, when calling `pad_sequences`, like this:

```
padded = pad_sequences(sequences, padding='post', maxlen=6)
```

The output from this will be:

```
[[ 2  3  4  5  6  0]
 [ 2  3  4  7  6  0]
 [ 3  8  5  2  0  0]
 [11 12 13 14 15  2]]
```

Now your padded sequences are all the same length, and there isn't too much padding. You have lost some words from your longest sentence, though, and they've been truncated from the beginning. What if you don't want to lose the words from the beginning and instead want them truncated from the *end* of the sentence? You can override the default behavior with the `truncating` parameter, as follows:

```
padded = pad_sequences(sequences, padding='post', maxlen=6, truncating='post')
```

The result of this will show that the longest sentence is now truncated at the end instead of the beginning:

```
[[ 2  3  4  5  6  0]
 [ 2  3  4  7  6  0]
 [ 3  8  5  2  0  0]
 [ 9 10 11 12 13 14]]
```



TensorFlow supports training using “ragged” (different-shaped) tensors, which is perfect for the needs of NLP. Using them is a bit more advanced than what we’re covering in this book, but once you’ve completed the introduction to NLP that’s provided in the next few chapters, you can explore the [documentation](#) to learn more.

Removing Stopwords and Cleaning Text

In the next section you’ll look at some real-world datasets, and you’ll find that there’s often text that you *don’t* want in your dataset. You may want to filter out so-called *stopwords* that are too common and don’t add any meaning, like “the,” “and,” and “but.” You may also encounter a lot of HTML tags in your text, and it would be good to have a clean way to remove them. Other things you might want to filter out include rude words, punctuation, or names. Later we’ll explore a dataset of tweets, which often have somebody’s user ID in them, and we’ll want to filter those out.

While every task is different based on your corpus of text, there are three main things that you can do to clean up your text programmatically.

The first is to strip out HTML tags. Fortunately, there's a library called BeautifulSoup that makes this straightforward. For example, if your sentences contain HTML tags such as
, they'll be removed by this code:

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(sentence)
sentence = soup.get_text()
```

A common way to remove stopwords is to have a stopwords list and to preprocess your sentences, removing instances of stopwords. Here's an abbreviated example:

```
stopwords = ["a", "about", "above", ... "yours", "yourself", "yourselves"]
```

A full stopwords list can be found in some of the online [examples](#) for this chapter.

Then, as you are iterating through your sentences, you can use code like this to remove the stopwords from your sentences:

```
words = sentence.split()
filtered_sentence = ""
for word in words:
    if word not in stopwords:
        filtered_sentence = filtered_sentence + word + " "
sentences.append(filtered_sentence)
```

Another thing you might consider is stripping out punctuation, which can fool a stopword remover. The one just shown looks for words surrounded by spaces, so a stopword immediately followed by a period or a comma won't be spotted.

Fixing this problem is easy with the translation functions provided by the Python string library. It also comes with a constant, `string.punctuation`, that contains a list of common punctuation marks, so to remove them from a word you can do the following:

```
import string
table = str.maketrans('', '', string.punctuation)
words = sentence.split()
filtered_sentence = ""
for word in words:
    word = word.translate(table)
    if word not in stopwords:
        filtered_sentence = filtered_sentence + word + " "
sentences.append(filtered_sentence)
```

Here, before filtering for stopwords, each word in the sentence has punctuation removed. So, if splitting a sentence gives you the word "it," it will be converted to "it" and then stripped out as a stopword. Note, however, that when doing this you might have to update your stopwords list. It's common for these lists to have abbreviated words and contractions like "you'll" in them. The translator will change "you'll" to "youll," and if you want to have that filtered out, you'll need to update your stopwords list to include it.

Following these three steps will give you a much cleaner set of text to use. But of course, every dataset will have its idiosyncrasies that you'll need to work with.

Working with Real Data Sources

Now that you've seen the basics of getting sentences, encoding them with a word index, and sequencing the results, you can take that to the next level by taking some well-known public datasets and using the tools Python provides to get them into a format where they can be easily sequenced. We'll start with one where a lot of the work has already been done for you in TensorFlow Datasets: the IMDb dataset. After that we'll get a bit more hands-on, processing a JSON-based dataset and a couple of comma-separated values (CSV) datasets with emotion data in them!

Getting Text from TensorFlow Datasets

We explored TFDS in [Chapter 4](#), so if you're stuck on any of the concepts in this section, you can take a quick look there. The goal behind TFDS is to make it as easy as possible to get access to data in a standardized way. It provides access to several text-based datasets; we'll explore `imdb_reviews`, a dataset of 50,000 labeled movie reviews from the Internet Movie Database (IMDb), each of which is determined to be positive or negative in sentiment.

This code will load the training split from the IMDb dataset and iterate through it, adding the text field containing the review to a list called `imdb_sentences`. Reviews are a tuple of the text and a label containing the sentiment of the review. Note that by wrapping the `tfds.load` call in `tfds.as_numpy` you ensure that the data will be loaded as strings, not tensors:

```
imdb_sentences = []
train_data = tfds.as_numpy(tfds.load('imdb_reviews', split="train"))
for item in train_data:
    imdb_sentences.append(str(item['text']))
```

Once you have the sentences, you can then create a tokenizer and fit it to them as before, as well as creating a set of sequences:

```
tokenizer = tf.keras.preprocessing.text.Tokenizer(num_words=5000)
tokenizer.fit_on_texts(imdb_sentences)
sequences = tokenizer.texts_to_sequences(imdb_sentences)
```

You can also print out your word index to inspect it:

```
print(tokenizer.word_index)
```

It's too large to show the entire index, but here are the top 20 words. Note that the tokenizer lists them in order of frequency in the dataset, so common words like “the,” “and,” and “a” are indexed:

```
{'the': 1, 'and': 2, 'a': 3, 'of': 4, 'to': 5, 'is': 6, 'br': 7, 'in': 8,
 'it': 9, 'i': 10, 'this': 11, 'that': 12, 'was': 13, 'as': 14, 'for': 15,
 'with': 16, 'movie': 17, 'but': 18, 'film': 19, 's': 20, ...}
```

These are stopwords, as described in the previous section. Having these present can impact your training accuracy because they're the most common words and they're nondistinct.

Also note that “br” is included in this list, because it's commonly used in this corpus as the
 HTML tag.

You can update the code to use BeautifulSoup to remove the HTML tags, add string translation to remove the punctuation, and remove stopwords from the given list as follows:

```
from bs4 import BeautifulSoup
import string

stopwords = ["a", ... , "yourselves"]

table = str.maketrans('', '', string.punctuation)

imdb_sentences = []
train_data = tfds.as_numpy(tfds.load('imdb_reviews', split="train"))
for item in train_data:
    sentence = str(item['text'].decode('UTF-8')).lower()
    soup = BeautifulSoup(sentence)
    sentence = soup.get_text()
    words = sentence.split()
    filtered_sentence = ""
    for word in words:
        word = word.translate(table)
        if word not in stopwords:
            filtered_sentence = filtered_sentence + word + " "
    imdb_sentences.append(filtered_sentence)

tokenizer = tf.keras.preprocessing.text.Tokenizer(num_words=25000)
tokenizer.fit_on_texts(imdb_sentences)
sequences = tokenizer.texts_to_sequences(imdb_sentences)
print(tokenizer.word_index)
```

Note that the sentences are converted to lowercase before processing because all the stopwords are stored in lowercase. When you print out your word index now, you'll see this:

```
{'movie': 1, 'film': 2, 'not': 3, 'one': 4, 'like': 5, 'just': 6, 'good': 7,
 'even': 8, 'no': 9, 'time': 10, 'really': 11, 'story': 12, 'see': 13,
 'can': 14, 'much': 15, ...}
```

You can see that this is much cleaner than before. There's always room to improve, however, and one thing I noted when looking at the full index was that some of the less common words toward the end were nonsensical. Often reviewers would

combine words, for example with a dash (“annoying-conclusion”) or a slash (“him/her”), and the stripping of punctuation would incorrectly turn these into a single word. You can avoid this with a bit of code that adds spaces around these characters, so I added the following immediately after the sentence was created:

```
sentence = sentence.replace(",", " , ")
sentence = sentence.replace(".", " . ")
sentence = sentence.replace("-", " - ")
sentence = sentence.replace("/", " / ")
```

This turned combined words like “him/her” into “him / her,” which then had the “/” stripped out and got tokenized into two words. This might lead to better training results later.

Now that you have a tokenizer for the corpus, you can encode your sentences. For example, the simple sentences we were looking at earlier in the chapter will come out like this:

```
sentences = [
    'Today is a sunny day',
    'Today is a rainy day',
    'Is it sunny today?'
]
sequences = tokenizer.texts_to_sequences(sentences)
print(sequences)

[[516, 5229, 147], [516, 6489, 147], [5229, 516]]
```

If you decode these, you’ll see that the stopwords are dropped and you get the sentences encoded as “today sunny day,” “today rainy day,” and “sunny today.”

If you want to do this in code, you can create a new dict with the reversed keys and values (i.e., for a key/value pair in the word index, make the value the key and the key the value) and do the lookup from that. Here’s the code:

```
reverse_word_index = dict(
    [(value, key) for (key, value) in tokenizer.word_index.items()])

decoded_review = ' '.join([reverse_word_index.get(i, '?') for i in sequences[0]])

print(decoded_review)
```

This will give the following result:

```
today sunny day
```

Using the IMDb subwords datasets

TFDS also contains a couple of preprocessed IMDb datasets using subwords. Here, you don’t have to break up the sentences by word; they have already been split up into subwords for you. Using subwords is a happy medium between splitting the corpus

into individual letters (relatively few tokens with low semantic meaning) and individual words (many tokens with high semantic meaning), and this approach can often be used very effectively to train a classifier for language. These datasets also include the encoders and decoders used to split and encode the corpus.

To access them, you can call `tfds.load` and pass it `imdb_reviews/subwords8k` or `imdb_reviews/subwords32k` like this:

```
(train_data, test_data), info = tfds.load(
    'imdb_reviews/subwords8k',
    split = (tfds.Split.TRAIN, tfds.Split.TEST),
    as_supervised=True,
    with_info=True
)
```

You can access the encoder on the `info` object like this. This will help you see the `vocab_size`:

```
encoder = info.features['text'].encoder
print('Vocabulary size: {}'.format(encoder.vocab_size))
```

This will output 8185 because the vocabulary in this instance is made up of 8,185 tokens. If you want to see the list of subwords, you can get it with the `encoder.subwords` property:

```
print(encoder.subwords)

['the_', ' ', ' ', ' ', 'a_', 'and_', 'of_', 'to_', 's_', 'is_', 'br', 'in_', 'I_',
 'that_', ...]
```

Some things you might notice here are that stopwords, punctuation, and grammar are all in the corpus, as are HTML tags like `
`. Spaces are represented by underscores, so the first token is the word “the.”

Should you want to encode a string, you can do so with the encoder like this:

```
sample_string = 'Today is a sunny day'

encoded_string = encoder.encode(sample_string)
print('Encoded string is {}'.format(encoded_string))
```

The output of this will be a list of tokens:

```
Encoded string is [6427, 4869, 9, 4, 2365, 1361, 606]
```

So, your five words are encoded into seven tokens. To see the tokens you can use the `subwords` property on the encoder, which returns an array. It's zero-based, so whereas “Tod” in “Today” was encoded as 6427, it's the 6,426th item in the array:

```
print(encoder.subwords[6426])
Tod
```

If you want to decode, you can use the `decode` method of the encoder:

```

encoded_string = encoder.encode(sample_string)

original_string = encoder.decode(encoded_string)
test_string = encoder.decode([6427, 4869, 9, 4, 2365, 1361, 606])

```

The latter lines will have an identical result because `encoded_string`, despite its name, is a list of tokens just like the one that is hardcoded on the next line.

Getting Text from CSV Files

While TFDS has lots of great datasets, it doesn't have everything, and often you'll need to manage loading the data yourself. One of the most common formats in which NLP data is available is CSV files. Over the next couple of chapters you'll use a CSV of Twitter data that I adapted from the open source [Sentiment Analysis in Text data-set](#). You will use two different datasets, one where the emotions have been reduced to "positive" or "negative" for binary classification and one where the full range of emotion labels is used. The structure of each is identical, so I'll just show the binary version here.

The Python `csv` library makes handling CSV files straightforward. In this case, the data is stored with two values per line. This first is a number (0 or 1) denoting if the sentiment is negative or positive. The second is a string containing the text.

The following code will read the CSV and do similar preprocessing to what we saw in the previous section. It adds spaces around the punctuation in compound words, uses `BeautifulSoup` to strip HTML content, and then removes all punctuation characters:

```

import csv
sentences=[]
labels=[]
with open('/tmp/binary-emotion.csv', encoding='UTF-8') as csvfile:
    reader = csv.reader(csvfile, delimiter=",")
    for row in reader:
        labels.append(int(row[0]))
        sentence = row[1].lower()
        sentence = sentence.replace(",", " , ")
        sentence = sentence.replace(".", " . ")
        sentence = sentence.replace("-", " - ")
        sentence = sentence.replace("/", " / ")
        soup = BeautifulSoup(sentence)
        sentence = soup.get_text()
        words = sentence.split()
        filtered_sentence = ""
        for word in words:
            word = word.translate(table)
            if word not in stopwords:
                filtered_sentence = filtered_sentence + word + " "
        sentences.append(filtered_sentence)

```

This will give you a list of 35,327 sentences.

Creating training and test subsets

Now that the text corpus has been read into a list of sentences, you'll need to split it into training and test subsets for training a model. For example, if you want to use 28,000 sentences for training with the rest held back for testing, you can use code like this:

```
training_size = 28000

training_sentences = sentences[0:training_size]
testing_sentences = sentences[training_size:]
training_labels = labels[0:training_size]
testing_labels = labels[training_size:]
```

Now that you have a training set, you need to create the word index from it. Here is the code to use the tokenizer to create a vocabulary with up to 20,000 words. We'll set the maximum length of a sentence at 10 words, truncate longer ones by cutting off the end, pad shorter ones at the end, and use "<OOV>":

```
vocab_size = 20000
max_length = 10
trunc_type='post'
padding_type='post'
oov_tok = "<OOV>"

tokenizer = Tokenizer(num_words=vocab_size, oov_token=oov_tok)
tokenizer.fit_on_texts(training_sentences)

word_index = tokenizer.word_index

training_sequences = tokenizer.texts_to_sequences(training_sentences)

training_padded = pad_sequences(training_sequences, maxlen=max_length,
                                padding=padding_type,
                                truncating=trunc_type)
```

You can inspect the results by looking at `training_sequences` and `training_padded`. For example, here we print the first item in the training sequence, and you can see how it's padded to a max length of 10:

```
print(training_sequences[0])
print(training_padded[0])

[18, 3257, 47, 4770, 613, 508, 951, 423]
[ 18 3257  47 4770  613 508 951 423   0   0]
```

You can also inspect the word index by printing it:

```
{ '<OOV>': 1, 'just': 2, 'not': 3, 'now': 4, 'day': 5, 'get': 6, 'no': 7,
  'good': 8, 'like': 9, 'go': 10, 'dont': 11, ... }
```

There are many words here you might want to consider getting rid of as stopwords, such as “like” and “dont.” It's always useful to inspect the word index.

Getting Text from JSON Files

Another very common format for text files is JavaScript Object Notation (JSON). This is an open standard file format used often for data interchange, particularly with web applications. It's human-readable and designed to use name/value pairs. As such, it's particularly well suited for labeled text. A quick search of Kaggle datasets for JSON yields over 2,500 results. Popular datasets such as the Stanford Question Answering Dataset (SQuAD), for example, are stored in JSON.

JSON has a very simple syntax, where objects are contained within braces as name/value pairs separated by a comma. For example, a JSON object representing my name would be:

```
{ "firstName" : "Laurence",  
  "lastName" : "Moroney" }
```

JSON also supports arrays, which are a lot like Python lists, and are denoted by the square bracket syntax. Here's an example:

```
[  
  { "firstName" : "Laurence",  
    "lastName" : "Moroney" },  
  { "firstName" : "Sharon",  
    "lastName" : "Agathon" }  
]
```

Objects can also contain arrays, so this is perfectly valid JSON:

```
[  
  { "firstName" : "Laurence",  
    "lastName" : "Moroney",  
    "emails": [ "lmoroney@gmail.com", "lmoroney@galactica.net" ]  
  },  
  { "firstName" : "Sharon",  
    "lastName" : "Agathon",  
    "emails": [ "sharon@galactica.net", "boomer@cylon.org" ]  
  }  
]
```

A smaller dataset that's stored in JSON and a lot of fun to work with is the News Headlines Dataset for Sarcasm Detection by [Rishabh Misra](#), available on [Kaggle](#). This dataset collects news headlines from two sources: *The Onion* for funny or sarcastic ones, and the *HuffPost* for normal headlines.

The file structure in the Sarcasm dataset is very simple:

```
{ "is_sarcastic": 1 or 0,  
  "headline": String containing headline,  
  "article_link": String containing link }
```

The dataset consists of about 26,000 items, one per line. To make it more readable in Python I've created a version that encloses these in an array so it can be read as a single list, which is used in the source code for this chapter.

Reading JSON files

Python's `json` library makes reading JSON files simple. Given that JSON uses name/value pairs, you can index the content based on the name. So, for example, for the Sarcasm dataset you can create a file handle to the JSON file, open it with the `json` library, have an iterable go through, read each field line by line, and get the data item using the name of the field.

Here's the code:

```
import json
with open("/tmp/sarcasm.json", 'r') as f:
    datastore = json.load(f)
    for item in datastore:
        sentence = item['headline'].lower()
        label = item['is_sarcastic']
        link = item['article_link']
```

This makes it simple to create lists of sentences and labels as you've done throughout this chapter, and then tokenize the sentences. You can also do preprocessing on the fly as you read a sentence, removing stopwords, HTML tags, punctuation, and more. Here's the complete code to create lists of sentences, labels, and URLs, while having the sentences cleaned of unwanted words and characters:

```
with open("/tmp/sarcasm.json", 'r') as f:
    datastore = json.load(f)

sentences = []
labels = []
urls = []
for item in datastore:
    sentence = item['headline'].lower()
    sentence = sentence.replace(",", " , ")
    sentence = sentence.replace(".", " . ")
    sentence = sentence.replace("-", " - ")
    sentence = sentence.replace("/", " / ")
    soup = BeautifulSoup(sentence)
    sentence = soup.get_text()
    words = sentence.split()
    filtered_sentence = ""
    for word in words:
        word = word.translate(table)
        if word not in stopwords:
            filtered_sentence = filtered_sentence + word + " "
    sentences.append(filtered_sentence)
    labels.append(item['is_sarcastic'])
    urls.append(item['article_link'])
```

As before, these can be split into training and test sets. If you want to use 23,000 of the 26,000 items in the dataset for training, you can do the following:

```
training_size = 23000

training_sentences = sentences[0:training_size]
testing_sentences = sentences[training_size:]
training_labels = labels[0:training_size]
testing_labels = labels[training_size:]
```

To tokenize the data and get it ready for training, you can follow the same approach as earlier. Here, we again specify a vocab size of 20,000 words, a maximum sequence length of 10 with truncation and padding at the end, and an OOV token of “<OOV>”:

```
vocab_size = 20000
max_length = 10
trunc_type='post'
padding_type='post'
oov_tok = "<OOV>"

tokenizer = Tokenizer(num_words=vocab_size, oov_token=oov_tok)
tokenizer.fit_on_texts(training_sentences)

word_index = tokenizer.word_index

training_sequences = tokenizer.texts_to_sequences(training_sentences)
padded = pad_sequences(training_sequences, padding='post')
print(word_index)
```

The output will be the whole index, in order of word frequency:

```
{ '<OOV>': 1, 'new': 2, 'trump': 3, 'man': 4, 'not': 5, 'just': 6, 'will': 7,
  'one': 8, 'year': 9, 'report': 10, 'area': 11, 'donald': 12, ... }
```

Hopefully the similar-looking code will help you see the pattern that you can follow when preparing text for neural networks to classify or generate. In the next chapter you’ll see how to build a classifier for text using embeddings, and in [Chapter 7](#) you’ll take that a step further, exploring recurrent neural networks. Then, in [Chapter 8](#), you’ll see how to further enhance the sequence data to create a neural network that can generate new text!

Summary

In earlier chapters you used images to build a classifier. Images, by definition, are highly structured. You know their dimension. You know the format. Text, on the other hand, can be far more difficult to work with. It’s often unstructured, can contain undesirable content such as formatting instructions, doesn’t always contain what you want, and often has to be filtered to remove nonsensical or irrelevant content. In this chapter you saw how to take text and convert it to numbers using word

tokenization, and then explored how to read and filter text in a variety of formats. Given these skills, you're now ready to take the next step and learn how *meaning* can be inferred from words—the first step in understanding natural language.

Making Sentiment Programmable Using Embeddings

In [Chapter 5](#) you saw how to take words and encode them into tokens. You then saw how to encode sentences full of words into sequences full of tokens, padding or truncating them as appropriate to end up with a well-shaped set of data that can be used to train a neural network. In none of that was there any type of modeling of the *meaning* of a word. While it's true that there's no absolute numeric encoding that could encapsulate meaning, there are relative ones. In this chapter you'll learn about them, and in particular the concept of *embeddings*, where vectors in high-dimensional space are created to represent words. The directions of these vectors can be learned over time based on the use of the words in the corpus. Then, when you're given a sentence, you can investigate the directions of the word vectors, sum them up, and from the overall direction of the summation establish the sentiment of the sentence as a product of its words.

In this chapter we'll explore how that works. Using the Sarcasm dataset from [Chapter 5](#), you'll build embeddings to help a model detect sarcasm in a sentence. You'll also see some cool visualization tools that help you understand how words in a corpus get mapped to vectors so you can see which words determine the overall classification.

Establishing Meaning from Words

Before we get into the higher-dimensional vectors for embeddings, let's try to visualize how meaning can be derived from numerics with some simple examples. Consider this: using the Sarcasm dataset from [Chapter 5](#), what would happen if you encoded all of the words that make up sarcastic headlines with positive numbers and those that make up realistic headlines with negative numbers?

A Simple Example: Positives and Negatives

Take, for example, this sarcastic headline from the dataset:

christian bale given neutered male statuette named oscar

Assuming that all words in our vocabulary start with a value of 0, we could add 1 to the values for each of the words in this sentence, and we would end up with this:

```
{ "christian" : 1, "bale" : 1, "given" : 1, "neutered": 1, "male" : 1,
  "statuette": 1, "named" : 1, "oscar": 1}
```



Note that this isn't the same as the *tokenization* of words that you did in the last chapter. You could consider replacing each word (e.g., "christian") with the token representing it that is encoded from the corpus, but I'll leave the words in for now to make it easier to read.

Then, in the next step, consider an ordinary headline, not a sarcastic one, like this:

gareth bale scores wonder goal against germany

Because this is a different sentiment we could instead subtract 1 from the current value of each word, so our value set would look like this:

```
{ "christian" : 1, "bale" : 0, "given" : 1, "neutered": 1, "male" : 1,
  "statuette": 1, "named" : 1, "oscar": 1, "gareth" : -1, "scores": -1,
  "wonder" : -1, "goal" : -1, "against" : -1, "germany" : -1}
```

Note that the sarcastic "bale" (from "christian bale") has been offset by the nonsarcastic "bale" (from "gareth bale"), so its score ends up as 0. Repeat this process thousands of times and you'll end up with a huge list of words from your corpus scored based on their usage.

Now imagine we want to establish the sentiment of this sentence:

neutered male named against germany, wins statuette!

Using our existing value set, we could look at the scores of each word and add them up. We would get a score of 2, indicating (because it's a positive number) that this is a sarcastic sentence.



For what it's worth, "bale" is used five times in the Sarcasm dataset, twice in a normal headline and three times in a sarcastic one, so in a model like this the word "bale" would be scored -1 across the whole dataset.

Going a Little Deeper: Vectors

Hopefully the previous example has helped you understand the mental model of establishing some form of *relative* meaning for a word, through its association with other words in the same “direction.” In our case, while the computer doesn’t understand the meanings of individual words, it can move labeled words from a known sarcastic headline in one direction (by adding 1) and labeled words from a known normal headline in another direction (by subtracting 1). This gives us a basic understanding of the meaning of the words, but it does lose some nuance.

What if we increased the dimensionality of the direction to try to capture some more information? For example, suppose we were to look at characters from the Jane Austen novel *Pride and Prejudice*, considering the dimensions of gender and nobility. We could plot the former on the x-axis and the latter on the y-axis, with the length of the vector denoting each character’s wealth (Figure 6-1).

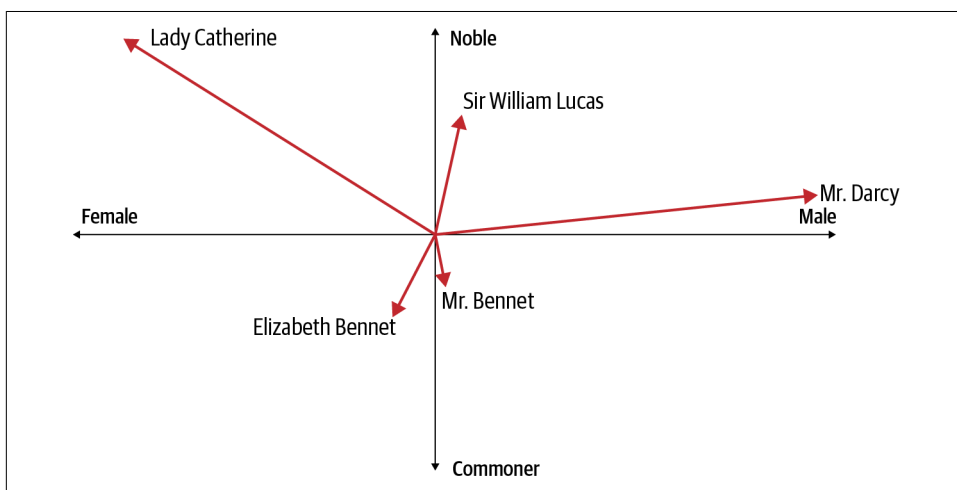


Figure 6-1. Characters in *Pride and Prejudice* as vectors

From an inspection of the graph, you can derive a fair amount of information about each character. Three of them are male. Mr. Darcy is extremely wealthy, but his nobility isn’t clear (he’s called “Mister,” unlike the less wealthy but apparently more noble Sir William Lucas). The other “Mister,” Mr. Bennet, is clearly not nobility and is struggling financially. Elizabeth Bennet, his daughter, is similar to him, but female. Lady Catherine, the other female character in our example, is nobility and incredibly wealthy. The romance between Mr. Darcy and Elizabeth causes tension—*prejudice* coming from the noble side of the vectors toward the less-noble.

As this example shows, by considering multiple dimensions we can begin to see real meaning in the words (here, character names). Again, we’re not talking about

concrete definitions, but more a *relative* meaning based on the axes and the relation between the vector for one word and the other vectors.

This leads us to the concept of an *embedding*, which is simply a vector representation of a word that is learned while training a neural network. We'll explore that next.

Embeddings in TensorFlow

As you've seen with Dense and Conv2D, `tf.keras` implements embeddings using a layer. This creates a lookup table that maps from an integer to an embedding table, the contents of which are the coefficients of the vector representing the word identified by that integer. So, in the *Pride and Prejudice* example from the previous section, the x and y coordinates would give us the embeddings for a particular character from the book. Of course, in a real NLP problem, we'll use far more than two dimensions. Thus, the direction of a vector in the vector space could be seen as encoding the “meaning” of a word, and words with similar vectors—i.e., pointing in roughly the same direction—could be considered related to that word.

The embedding layer will be initialized randomly—that is, the coordinates of the vectors will be completely random to start and will be learned during training using backpropagation. When training is complete, the embeddings will roughly encode similarities between words, allowing us to identify words that are somewhat similar based on the direction of the vectors for those words.

This is all quite abstract, so I think the best way to understand how to use embeddings is to roll your sleeves up and give them a try. Let's start with a sarcasm detector using the Sarcasm dataset from [Chapter 5](#).

Building a Sarcasm Detector Using Embeddings

In [Chapter 5](#) you loaded and did some preprocessing on a JSON dataset called the News Headlines Dataset for Sarcasm Detection (Sarcasm, for short). By the time you were done you had lists of training and testing data and labels. These can be converted to Numpy format, used by TensorFlow for training, with code like this:

```
import numpy as np
training_padded = np.array(training_padded)
training_labels = np.array(training_labels)
testing_padded = np.array(testing_padded)
testing_labels = np.array(testing_labels)
```

These were created using a tokenizer with a specified maximum vocabulary size and out-of-vocabulary token:

```
tokenizer = Tokenizer(num_words=vocab_size, oov_token=oov_tok)
```

To initialize the embedding layer, you'll need the vocab size and a specified number of embedding dimensions:

```
tf.keras.layers.Embedding(vocab_size, embedding_dim),
```

This will initialize an array with `embedding_dim` points for each word. So, for example, if `embedding_dim` is 16, every word in the vocabulary will be assigned a 16-dimensional vector.

Over time, the dimensions will be learned through backpropagation as the network learns by matching the training data to its labels.

An important next step is then feeding the output of the embedding layer into a dense layer. The easiest way to do this, similar to how you would when using a convolutional neural network, is to use pooling. In this instance, the dimensions of the embeddings are averaged out to produce a fixed-length output vector.

As an example, consider this model architecture:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(10000, 16),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(loss='binary_crossentropy',
              optimizer='adam', metrics=['accuracy'])
```

Here an embedding layer is defined, and it's given the vocab size (10000) and an embedding dimension of 16. Let's take a look at the number of trainable parameters in the network, using `model.summary()`:

Model: "sequential_2"

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, None, 16)	160000
global_average_pooling1d_2 ((None, 16)		0
dense_4 (Dense)	(None, 24)	408
dense_5 (Dense)	(None, 1)	25
Total params: 160,433		
Trainable params: 160,433		
Non-trainable params: 0		

As the embedding has a 10,000-word vocabulary, and each word will be a vector in 16 dimensions, the total number of trainable parameters will be 160,000.

The average pooling layer has 0 trainable parameters, as it's just averaging the parameters in the embedding layer before it, to get a single 16-value vector.

This is then fed into the 24-neuron dense layer. Remember that a dense neuron effectively calculates using weights and biases, so it will need to learn $(24 \times 16) + 16 = 408$ parameters.

The output of this layer is then passed to the final single-neuron layer, where there will be $(1 \times 24) + 1 = 25$ parameters to learn.

If we train this model, we'll get a pretty decent accuracy of 99+% after 30 epochs—but our validation accuracy will only be about 81% (Figure 6-2).

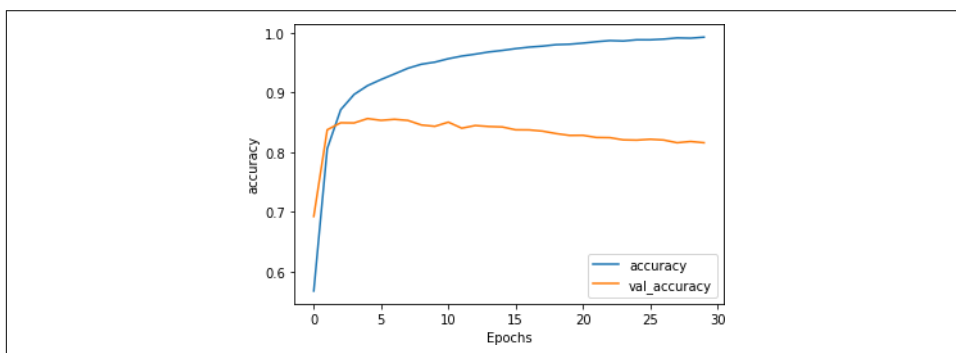


Figure 6-2. Training accuracy versus validation accuracy

That might seem to be a reasonable curve given that the validation data likely contains many words that aren't present in the training data. However, if you examine the loss curves for training versus validation over the 30 epochs, you'll see a problem. Although you would expect to see that the training accuracy is higher than the validation accuracy, a clear indicator of overfitting is that while the validation accuracy is dropping a little over time (in Figure 6-2), its loss is increasing sharply as shown in Figure 6-3.

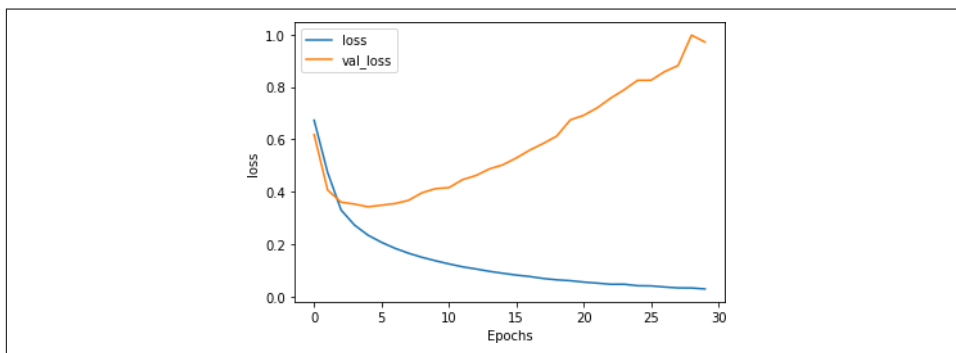


Figure 6-3. Training loss versus validation loss

Overfitting like this is common with NLP models due to the somewhat unpredictable nature of language. In the next sections, we'll look at how to reduce this effect using a number of techniques.

Reducing Overfitting in Language Models

Overfitting happens when the network becomes overspecialized to the training data, and one part of this is that it has become very good at matching patterns in “noisy” data in the training set that doesn't exist anywhere else. Because this particular noise isn't present in the validation set, the better the network gets at matching it, the worse the loss of the validation set will be. This can result in the escalating loss that you saw in [Figure 6-3](#). In this section, we'll explore several ways to generalize the model and reduce overfitting.

Adjusting the learning rate

Perhaps the biggest factor that can lead to overfitting is if the learning rate of your optimizer is too high. This means that the network learns *too quickly*. For this example, the code to compile the model was as follows:

```
model.compile(loss='binary_crossentropy',
              optimizer='adam', metrics=['accuracy'])
```

The optimizer is simply declared as `adam`, which invokes the Adam optimizer with default parameters. This optimizer, however, supports multiple parameters, including learning rate. You can change the code to this:

```
adam = tf.keras.optimizers.Adam(learning_rate=0.0001,
                                beta_1=0.9, beta_2=0.999, amsgrad=False)

model.compile(loss='binary_crossentropy',
              optimizer=adam, metrics=['accuracy'])
```

where the default value for learning rate, which is typically 0.001, has been reduced by 90%, to 0.0001. The `beta_1` and `beta_2` values stay at their defaults, as does `amsgrad`. `beta_1` and `beta_2` must be between 0 and 1, and typically both are close to 1. Amsgrad is an alternative implementation of the Adam optimizer, introduced in the paper “[On the Convergence of Adam and Beyond](#)” by Sashank Reddi, Satyen Kale, and Sanjiv Kumar.

This much lower learning rate has a profound impact on the network. [Figure 6-4](#) shows the accuracy of the network over 100 epochs. The lower learning rate can be seen in the first 10 epochs or so, where it appears that the network isn't learning, before it “breaks out” and starts to learn quickly.

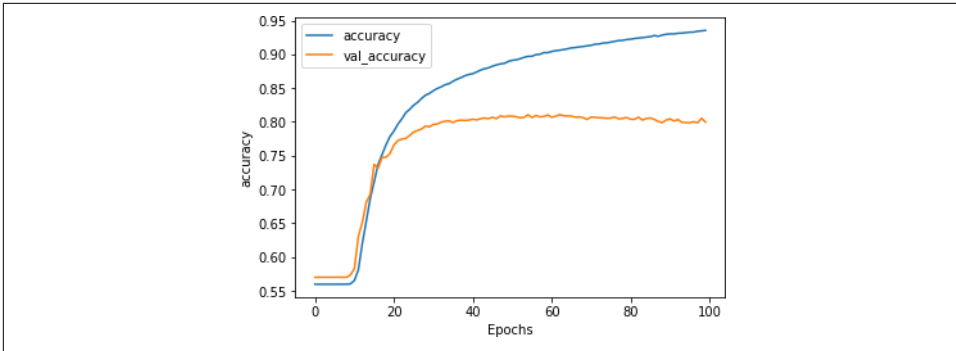


Figure 6-4. Accuracy with a lower learning rate

Exploring the loss (as illustrated in [Figure 6-5](#)) we can see that even while the accuracy wasn't going up for the first few epochs, the loss was going down, so you could be confident that the network would start to learn eventually if you were watching it epoch by epoch.

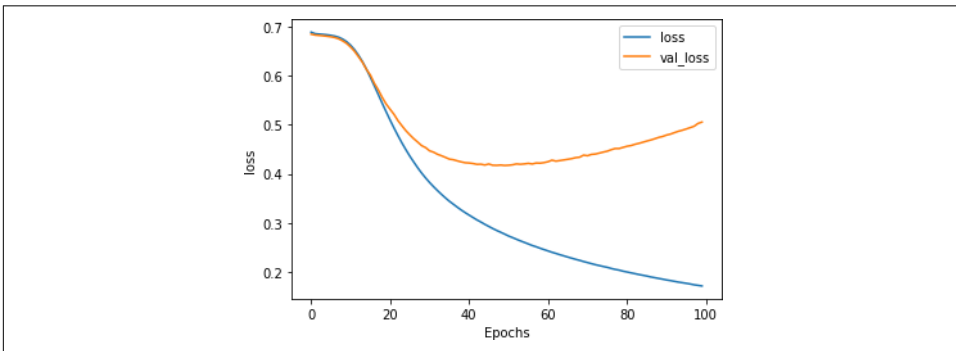


Figure 6-5. Loss with a lower learning rate

And while the loss does start to show the same curve of overfitting that you saw in [Figure 6-3](#), note that it happens much later, and at a much lower rate. By epoch 30 the loss is at about 0.45, whereas with the higher learning rate in [Figure 6-3](#) it was more than double that amount. And while it takes the network longer to get to a good accuracy rate, it does so with less loss, so you can be more confident in the results. With these hyperparameters, the loss on the validation set started to increase at about epoch 60, at which point the training set had 90% accuracy and the validation set about 81%, showing that we have quite an effective network.

Of course, it's easy to just tweak the optimizer and then declare victory, but there are a number of other methods you can use to improve your model, which you'll see in the next few sections. For these, I reverted back to using the default Adam optimizer

so the effects of tweaking the learning rate don't hide the benefits offered by these other techniques.

Exploring vocabulary size

The Sarcasm dataset deals with words, so if you explore the words in the dataset, and in particular their frequency, you might get a clue that helps fix the overfitting issue.

The tokenizer gives you a way of doing this with its `word_counts` property. If you were to print it you'd see something like this, an `OrderedDict` containing tuples of word and word count:

```
wc=tokenizer.word_counts
print(wc)

OrderedDict([('former', 75), ('versace', 1), ('store', 35), ('clerk', 8),
             ('sues', 12), ('secret', 68), ('black', 203), ('code', 16),...])
```

The order of the words is determined by their order of appearance within the dataset. If you look at the first headline in the training set, it's a sarcastic one about a former Versace store clerk. Stopwords have been removed; otherwise you'd see a high volume of words like "a" and "the."

Given that it's an `OrderedDict`, you can sort it into descending order of word volume:

```
from collections import OrderedDict
newlist = (OrderedDict(sorted(wc.items(), key=lambda t: t[1], reverse=True)))
print(newlist)

OrderedDict([('new', 1143), ('trump', 966), ('man', 940), ('not', 555), ('just',
430), ('will', 427), ('one', 406), ('year', 386), ...])
```

If you want to plot this, you can iterate through each item in the list and make the *x* value the ordinal of where you are (1 for the first item, 2 for the second item, etc.). The *y* value will then be `newlist[item]`. This can then be plotted with `matplotlib`. Here's the code:

```
xs=[]
ys=[]
curr_x = 1
for item in newlist:
    xs.append(curr_x)
    curr_x=curr_x+1
    ys.append(newlist[item])

plt.plot(xs,ys)
plt.show()
```

The result is shown in [Figure 6-6](#).

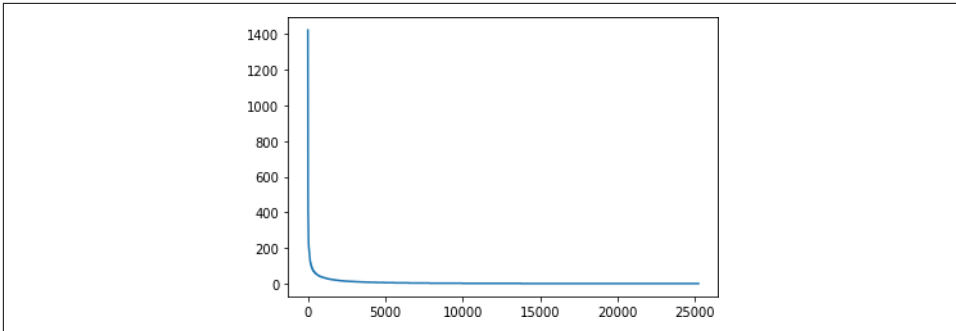


Figure 6-6. Exploring the frequency of words

This “hockey stick” curve shows us that very few words are used many times, whereas most words are used very few times. But every word is effectively weighted equally because every word has an “entry” in the embedding. Given that we have a relatively large training set in comparison with the validation set, we’re ending up in a situation where there are many words present in the training set that aren’t present in the validation set.

You can zoom in on the data by changing the axis of the plot just before calling `plt.show`. For example, to look at the volume of words 300 to 10,000 on the x-axis with the scale from 0 to 100 on the y-axis, you can use this code:

```
plt.plot(xs,ys)
plt.axis([300,10000,0,100])
plt.show()
```

The result is in [Figure 6-7](#).

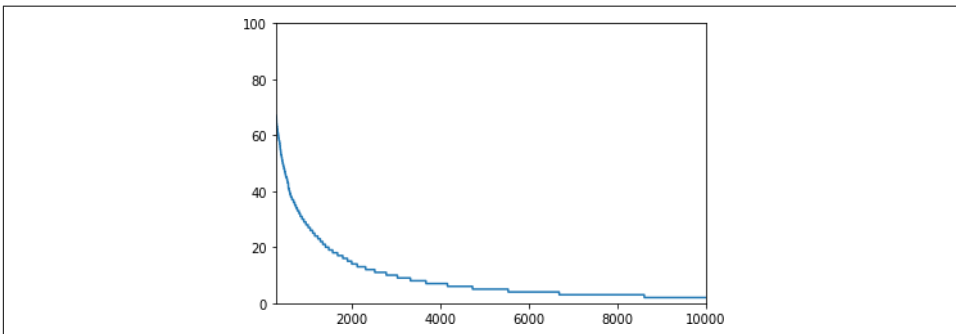


Figure 6-7. Frequency of words 300–10,000

While there are over 20,000 words in the corpus, the code is set up to only train for 10,000. But if we look at the words in positions 2,000–10,000, which is over 80% of our vocabulary, we see that they’re each used less than 20 times in the entire corpus.

This could explain the overfitting. Now consider what happens if you change the vocab size to two thousand and retrain. **Figure 6-8** shows the accuracy metrics. Now the training set accuracy is ~82% and the validation accuracy is about 76%. They're closer to each other and not diverging, which is a good sign that we've gotten rid of most of the overfitting.

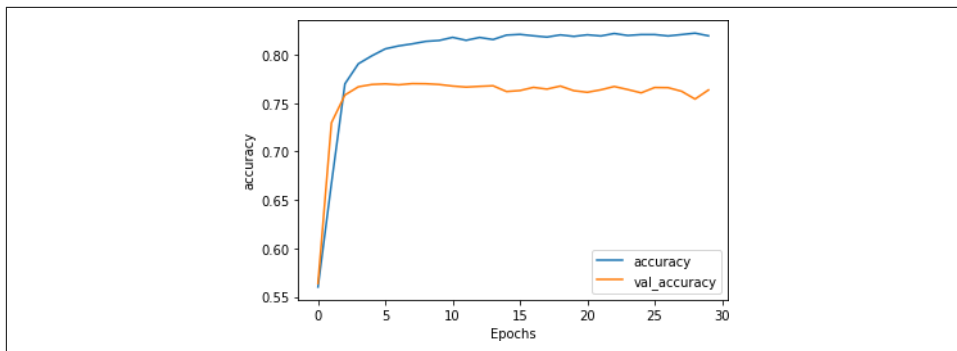


Figure 6-8. Accuracy with a two thousand-word vocabulary

This is somewhat reinforced by the loss plot in **Figure 6-9**. The loss on the validation set is rising, but much slower than before, so reducing the size of the vocabulary to prevent the training set from overfitting on low-frequency words that were possibly only present in the training set appears to have worked.

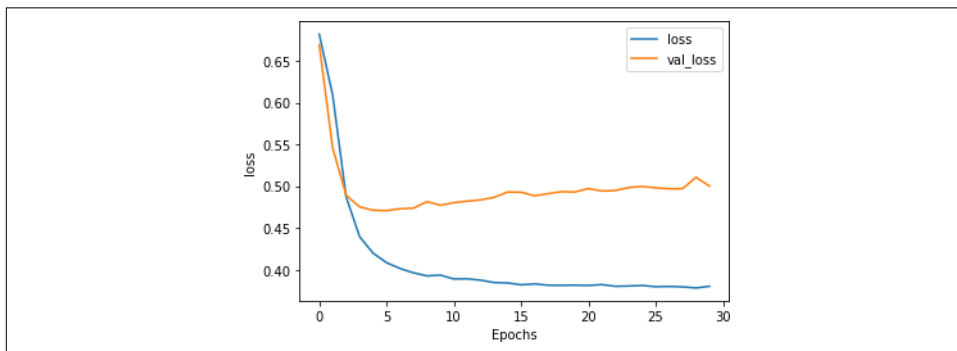


Figure 6-9. Loss with a two thousand-word vocabulary

It's worth experimenting with different vocab sizes, but remember that you can also have too small a vocab size and overfit to that. You'll need to find a balance. In this case, my choice of taking words that appear 20 times or more was purely arbitrary.

Exploring embedding dimensions

For this example, an embedding dimension of 16 was arbitrarily chosen. In this instance words are encoded as vectors in 16-dimensional space, with their directions

indicating their overall meaning. But is 16 a good number? With only two thousand words in our vocabulary it might be on the high side, leading to a high degree of sparseness of direction.

Best practice for embedding size is to have it be the fourth root of the vocab size. The fourth root of 2,000 is 6.687, so let's explore what happens if we change the embedding dimension to 7 and retrain the model for 100 epochs.

You can see the result on the accuracy in [Figure 6-9](#). The training set's accuracy stabilized at about 83% and the validation set's at about 77%. Despite some jitters, the lines are pretty flat, showing that the model has converged. This isn't much different from the results in [Figure 6-6](#), but reducing the embedding dimensionality allows the model to train over 30% faster.

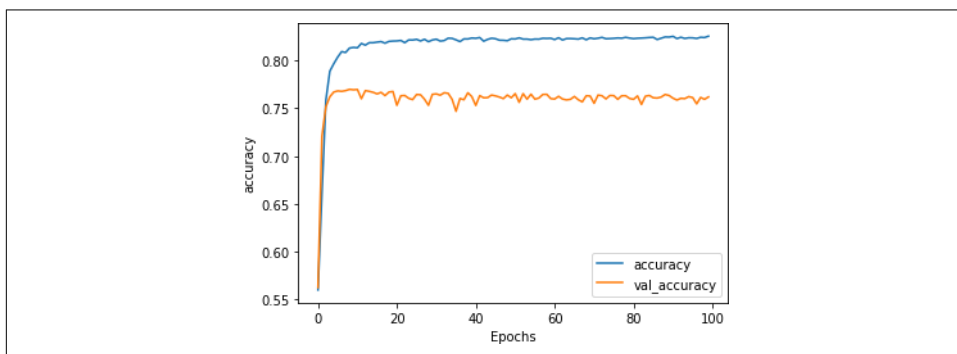


Figure 6-10. Training versus validation accuracy for seven dimensions

[Figure 6-11](#) shows the loss in training and validation. While it initially appeared that the loss was climbing at about epoch 20, it soon flattened out. Again, a good sign!

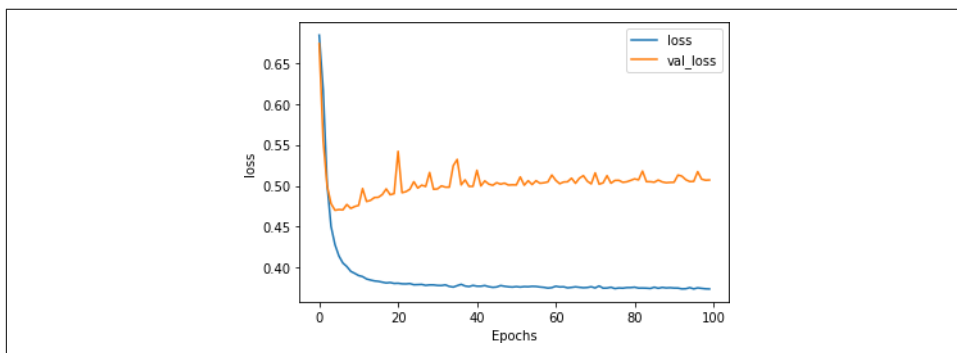


Figure 6-11. Training versus validation loss for seven dimensions

Now that the dimensionality has been reduced, we can do a bit more tweaking of the model architecture.

Exploring the model architecture

After the optimizations in the previous sections, the model architecture now looks like this:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(2000, 7),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(loss='binary_crossentropy',
              optimizer='adam', metrics=['accuracy'])
```

One thing that jumps to mind is the dimensionality—the `GlobalAveragePooling1D` layer is now emitting just seven dimensions, but they’re being fed into a dense layer of 24 neurons, which is overkill. Let’s explore what happens when this is reduced to just eight neurons and trained for one hundred epochs.

You can see the training versus validation accuracy in [Figure 6-12](#). When compared to [Figure 6-7](#), where 24 neurons were used, the overall result is quite similar, but the fluctuations have been smoothed out (visible in the lines being less jaggy). It’s also somewhat faster to train.

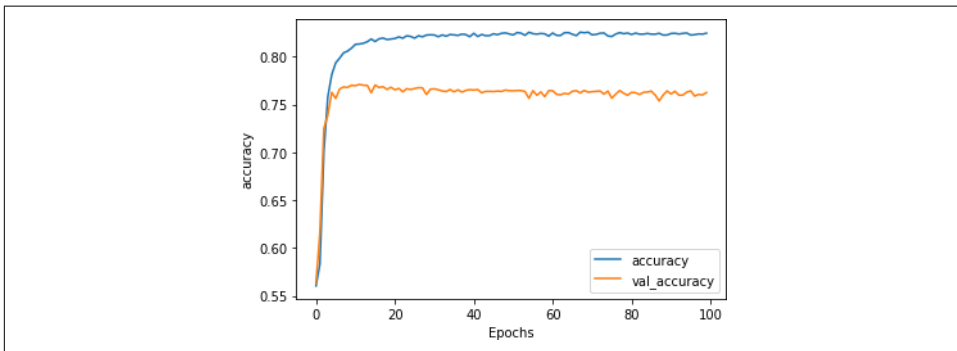


Figure 6-12. Reduced dense architecture accuracy results

Similarly, the loss curves in [Figure 6-13](#) show similar results, but with the jagginess reduced.

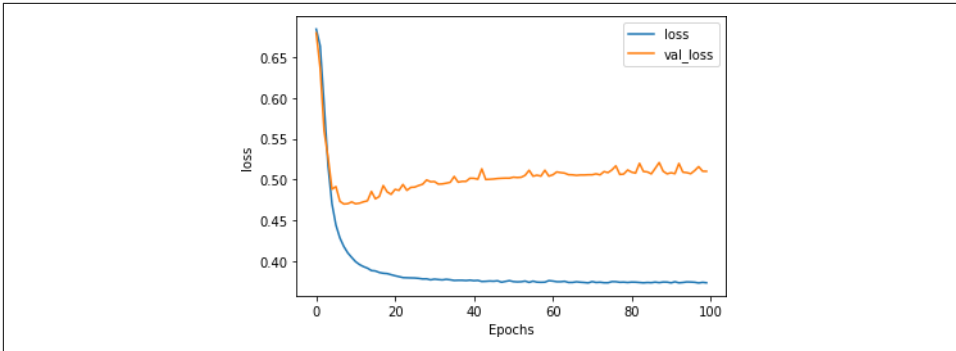


Figure 6-13. Reduced dense architecture loss results

Using dropout

A common technique for reducing overfitting is to add dropout to a dense neural network. We explored this for convolutional neural networks back in [Chapter 3](#). It's tempting to go straight to that to see its effects on overfitting, but in this case I wanted to wait until the vocabulary size, embedding size, and architecture complexity had been addressed. Those changes can often have a much larger impact than using dropout, and we've already seen some nice results.

Now that our architecture has been simplified to have only eight neurons in the middle dense layer, the effect of dropout may be minimized but let's explore it anyway. Here's the updated code for the model architecture adding a dropout of 0.25 (which equates to two of our eight neurons):

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dropout(.25),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

[Figure 6-14](#) shows the accuracy results when trained for one hundred epochs.

This time we see that the training accuracy is climbing above its previous threshold, while the validation accuracy is slowly dropping. This is a sign that we are entering overfitting territory again. This is confirmed by exploring the loss curves in [Figure 6-15](#).

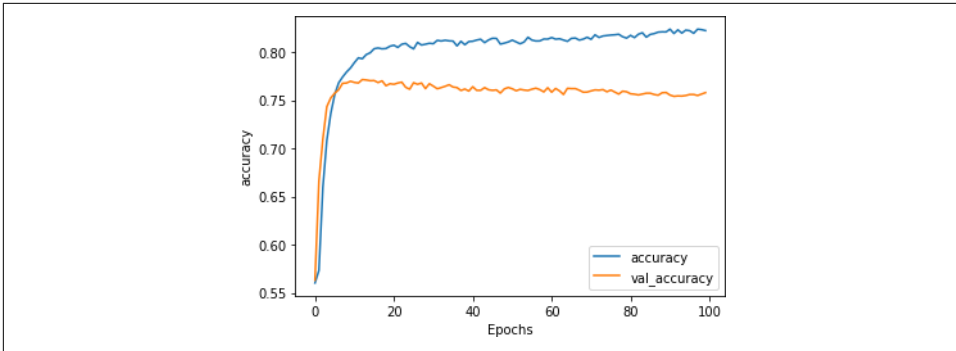


Figure 6-14. Accuracy with added dropout

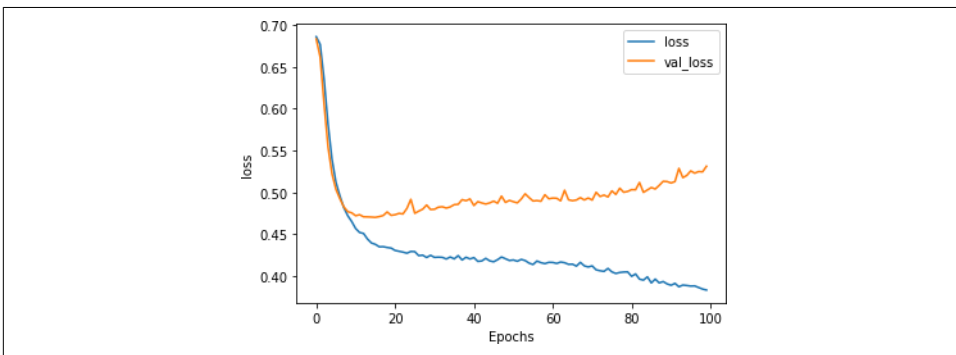


Figure 6-15. Loss with added dropout

Here you can see that the model is heading back to its previous pattern of increasing validation loss over time. It's not nearly as bad as before, but it's heading in the wrong direction.

In this case, when there were very few neurons, introducing dropout probably wasn't the right idea. It's still good to have this tool in your arsenal though, so be sure to keep it in mind for more sophisticated architectures than this one.

Using regularization

Regularization is a technique that helps prevent overfitting by reducing the polarization of weights. If the weights on some of the neurons are too heavy, regularization effectively punishes them. Broadly speaking, there are two types of regularization: *L1* and *L2*.

L1 regularization is often called *lasso* (least absolute shrinkage and selection operator) regularization. It effectively helps us ignore the zero or close-to-zero weights when calculating a result in a layer.

L2 regularization is often called *ridge* regression because it pushes values apart by taking their squares. This tends to amplify the differences between nonzero values and zero or close-to-zero ones, creating a ridge effect.

The two approaches can also be combined into what is sometimes called *elastic* regularization.

For NLP problems like the one we're considering, L2 is most commonly used. It can be added as an attribute to the Dense layer using the `kernel_regularizer` property, and takes a floating-point value as the regularization factor. This is another hyperparameter that you can experiment with to improve your model!

Here's an example:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(8, activation='relu',
        kernel_regularizer = tf.keras.regularizers.l2(0.01)),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

The impact of adding regularization in a simple model like this isn't particularly large, but it does smooth out our training loss and validation loss somewhat. It might be overkill for this scenario, but like dropout, it's a good idea to understand how to use regularization to prevent your model from getting overspecialized.

Other optimization considerations

While the modifications we've made have given us a much improved model with less overfitting, there are other hyperparameters that you can experiment with. For example, we chose to make the maximum sentence length one hundred, but this was purely arbitrary and probably not optimal. It's a good idea to explore the corpus and see what a better sentence length might be. Here's a snippet of code that looks at the sentences and plots the lengths of each one, sorted from low to high:

```
xs=[]
ys=[]
current_item=1
for item in sentences:
    xs.append(current_item)
    current_item=current_item+1
    ys.append(len(item))
newys = sorted(ys)

import matplotlib.pyplot as plt
plt.plot(xs,newys)
plt.show()
```

The results of this are shown in [Figure 6-16](#).

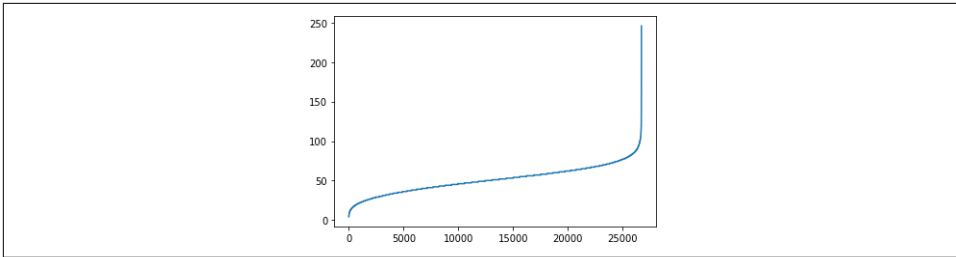


Figure 6-16. Exploring sentence length

Less than 200 sentences in the total corpus of 26,000+ have a length of 100 words or greater, so by choosing this as the max length, we’re introducing a lot of padding that isn’t necessary, and affecting the model’s performance. Reducing it to 85 would still keep 26,000 of the sentences (99%+) without any padding at all.

Using the Model to Classify a Sentence

Now that you’ve created the model, trained it, and optimized it to remove a lot of the problems that caused the overfitting, the next step is to run the model and inspect its results. To do this, create an array of new sentences. For example:

```
sentences = ["granny starting to fear spiders in the garden might be real",
             "game of thrones season finale showing this sunday night",
             "TensorFlow book will be a best seller"]
```

These can then be encoded using the same tokenizer that was used when creating the vocabulary for training. It’s important to use that because it has the tokens for the words that the network was trained on!

```
sequences = tokenizer.texts_to_sequences(sentences)
print(sequences)
```

The output of the print statement will be the sequences for the preceding sentences:

```
[[1, 816, 1, 691, 1, 1, 1, 1, 300, 1, 90],
 [111, 1, 1044, 173, 1, 1, 1, 1463, 181],
 [1, 234, 7, 1, 1, 46, 1]]
```

There are a lot of 1 tokens here (“<OOV>”), because stopwords like “in” and “the” have been removed from the dictionary, and words like “granny” and “spiders” don’t appear in the dictionary.

Before you can pass the sequences to the model, they’ll need to be in the shape that the model expects—that is, the desired length. You can do this with `pad_sequences` in the same way you did when training the model:

```
padded = pad_sequences(sequences, maxlen=max_length,
                       padding=padding_type, truncating=trunc_type)
print(padded)
```

This will output the sentences as sequences of length 100, so the output for the first sequence will be:

```
[ 1 816 1 691 1 1 1 1 300 1 90 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0]
```

It was a very short sentence!

Now that the sentences have been tokenized and padded to fit the model's expectations for the input dimensions, it's time to pass them to the model and get predictions back. This is as easy as doing this:

```
print(model.predict(padded))
```

The results will be passed back as a list and printed, with high values indicating likely sarcasm. Here are the results for our sample sentences:

```
[[0.7194135 ]
 [0.02041999]
 [0.13156283]]
```

The high score for the first sentence ("granny starting to fear spiders in the garden might be real"), despite it having a lot of stopwords and being padded with a lot of zeros, indicates that there is a high level of sarcasm here. The other two sentences scored much lower, indicating a lower likelihood of sarcasm within.

Visualizing the Embeddings

To visualize embeddings you can use a tool called the **Embedding Projector**. It comes preloaded with many existing datasets, but in this section you'll see how to take the data from the model you've just trained and visualize it using this tool.

First, you'll need a function to reverse the word index. It currently has the word as the token and the key as the value, but this needs to be inverted so we have word values to plot on the projector. Here's the code to do this:

```
reverse_word_index = dict([(value, key)
    for (key, value) in word_index.items()])
```

You'll also need to extract the weights of the vectors in the embeddings:

```
e = model.layers[0]
weights = e.get_weights()[0]
print(weights.shape)
```

The output of this will be (2000,7) if you followed the optimizations in this chapter—we used a 2,000 word vocabulary, and 7 dimensions for the embedding. If you want to explore a word and its vector details, you can do so with code like this:

```
print(reverse_word_index[2])
print(weights[2])
```

This will produce the following output:

```
new
[ 0.8091359  0.54640186 -0.9058702 -0.94764805 -0.8809764 -0.70225513
 0.86525863]
```

So, the word “new” is represented by a vector with those seven coefficients on its axes.

The Embedding Projector uses two tab-separated values (TSV) files, one for the vector dimensions and one for metadata. This code will generate them for you:

```
import io

out_v = io.open('vecs.tsv', 'w', encoding='utf-8')
out_m = io.open('meta.tsv', 'w', encoding='utf-8')
for word_num in range(1, vocab_size):
    word = reverse_word_index[word_num]
    embeddings = weights[word_num]
    out_m.write(word + "\n")
    out_v.write('\t'.join([str(x) for x in embeddings]) + "\n")
out_v.close()
out_m.close()
```

If you are using Google Colab, you can download the TSV files with the following code or from the Files pane:

```
try:
    from google.colab import files
except ImportError:
    pass
else:
    files.download('vecs.tsv')
    files.download('meta.tsv')
```

Once you have them you can press the Load button on the projector to visualize the embeddings, as shown in [Figure 6-17](#).

Use the vectors and meta TSV files where recommended in the resulting dialog, and then click Sphereize Data on the projector. This will cause the words to be clustered on a sphere and will give you a clear visualization of the binary nature of this classifier. It’s only been trained on sarcastic and nonsarcastic sentences, so words tend to cluster toward one label or another ([Figure 6-18](#)).

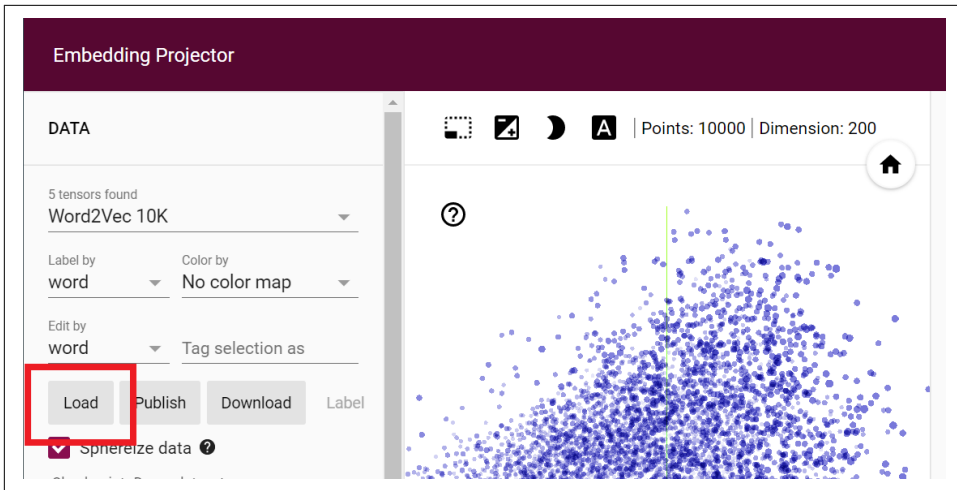


Figure 6-17. Using the Embeddings Projector

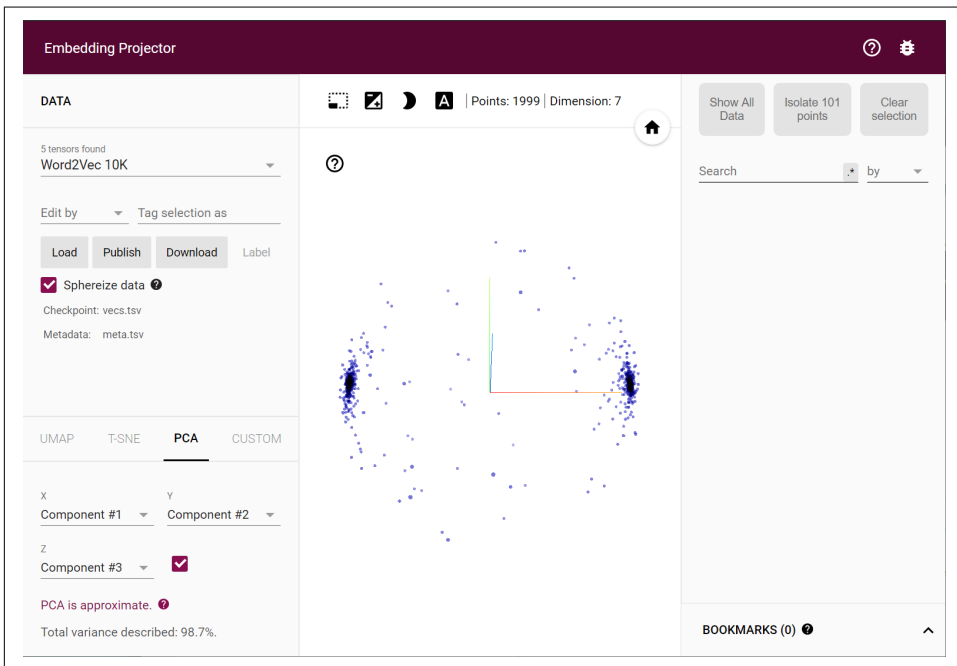


Figure 6-18. Visualizing the sarcasm embeddings

Screenshots don't do it justice; you should try it for yourself! You can rotate the center sphere and explore the words on each "pole" to see the impact they have on the overall classification. You can also select words and show related ones in the right-hand pane. Have a play and experiment.

Using Pretrained Embeddings from TensorFlow Hub

An alternative to training your own embeddings is to use ones that have been pre-trained and packaged into Keras layers for you. There are many of these on [TensorFlow Hub](#) that you can explore. One thing to note is that they can also contain the tokenization logic for you, so you don't have to handle the tokenization, sequencing, and padding yourself as you have been doing so far.

TensorFlow Hub is preinstalled in Google Colab, so the code in this chapter will work as is. If you want to install it as a dependency on your machine, you'll need to follow the [instructions](#) to install the latest version.

For example, with the Sarcasm data, instead of all the logic for tokenization, vocab management, sequencing, padding, etc., you could just do something like this once you have your full set of sentences and labels. First, split them into training and test sets:

```
training_size = 24000
training_sentences = sentences[0:training_size]
testing_sentences = sentences[training_size:]
training_labels = labels[0:training_size]
testing_labels = labels[training_size:]
```

Once you have these, you can download a pretrained layer from TensorFlow Hub like this:

```
import tensorflow_hub as hub

hub_layer = hub.KerasLayer(
    "https://tfhub.dev/google/tf2-preview/gnews-swivel-20dim/1",
    output_shape=[20], input_shape=[],
    dtype=tf.string, trainable=False
)
```

This takes the embeddings from the Swivel dataset, trained on 130 GB of Google News. Using this layer will encode your sentences, tokenize them, use the words from them with the embeddings learned as part of Swivel, and then *encode your sentences into a single embedding*. It's worth remembering that final part. The technique we've been using to date is to just use the word encodings and classify the content based on all of them. When using a layer like this, you're getting the full sentence aggregated into a new encoding.

You can then create a model architecture by using this layer instead of an embedding one. Here's a simple model that uses it:

```
model = tf.keras.Sequential([
    hub_layer,
    tf.keras.layers.Dense(16, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

adam = tf.keras.optimizers.Adam(learning_rate=0.0001, beta_1=0.9,
                                beta_2=0.999, amsgrad=False)

model.compile(loss='binary_crossentropy', optimizer=adam,
              metrics=['accuracy'])
```

This model will rapidly reach peak accuracy in training, and will not overfit as much as we saw previously. The accuracy over 50 epochs shows training and validation very much in step with each other (Figure 6-19).

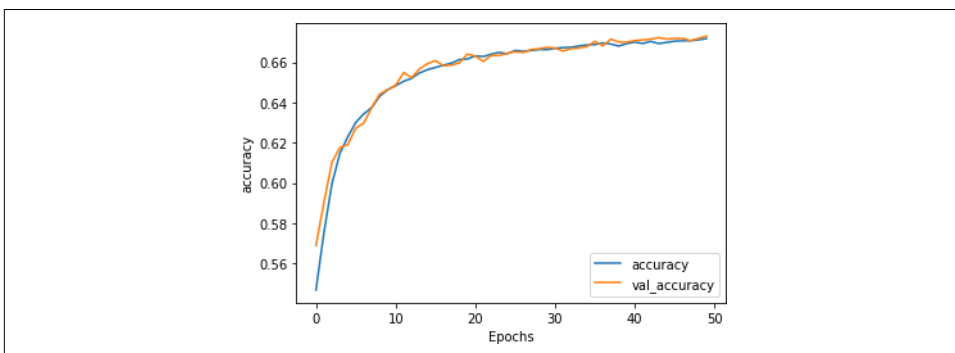


Figure 6-19. Accuracy metrics using Swivel embeddings

The loss values are also in step, showing that we are fitting very nicely (Figure 6-20).

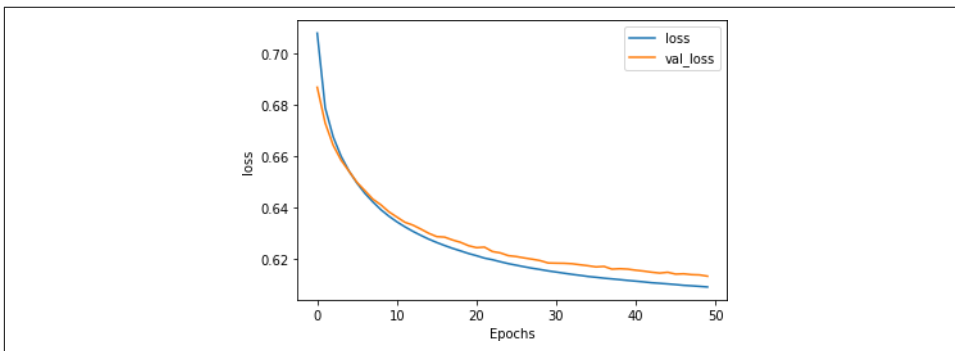


Figure 6-20. Loss metrics using Swivel embeddings

It is worth noting, however, that the overall accuracy (at about 67%) is quite low, considering a coin flip would have a 50% chance of getting it right! This is caused by the encoding of all of the word-based embeddings into a sentence-based one—in the case of sarcastic headlines, it appears that individual words can have a huge effect on the classification (see [Figure 6-18](#)). So, while using pretrained embeddings can make for much faster training with less overfitting, you should also understand what it is that they're useful for, and that they may not always be best for your scenario.

Summary

In this chapter, you built your first model to understand sentiment in text. It did it by taking the tokenized text from [Chapter 5](#) and mapping it to vectors. Then, using backpropagation, it learned the appropriate “direction” for each vector based on the label for the sentence containing it. Finally, it was able to use all of the vectors for a collection of words to build up an idea of the sentiment within the sentence. You also explored ways to optimize your model to avoid overfitting and saw a neat visualization of the final vectors representing your words. While this was a nice way to classify sentences, it simply treated each sentence as a bunch of words. There was no inherent sequence involved, and as the order of appearance of words is very important in determining the real meaning of a sentence, it's a good idea to see if we can improve our models by taking sequence into account. We'll explore that in the next chapter with the introduction of a new layer type—a *recurrent* layer, which is the foundation of recurrent neural networks. You'll also see another pretrained embedding, called GloVe, which allows you to use word-based embeddings in a transfer learning scenario.

Recurrent Neural Networks for Natural Language Processing

In [Chapter 5](#) you saw how to tokenize and sequence text, turning sentences into tensors of numbers that could then be fed into a neural network. You then extended that in [Chapter 6](#) by looking at embeddings, a way to have words with similar meanings cluster together to enable the calculation of sentiment. This worked really well, as you saw by building a sarcasm classifier. But there's a limitation to that, namely in that sentences aren't just bags of words—often the *order* in which the words appear will dictate their overall meaning. Adjectives can add to or change the meaning of the nouns they appear beside. For example, the word “blue” might be meaningless from a sentiment perspective, as might “sky,” but when you put them together to get “blue sky” there's a clear sentiment there that's usually positive. And some nouns may qualify others, such as “rain cloud,” “writing desk,” “coffee mug.”

To take sequences like this into account, an additional approach is needed, and that is to factor *recurrence* into the model architecture. In this chapter you'll look at different ways of doing this. We'll explore how sequence information can be learned, and how this information can be used to create a type of model that is better able to understand text: the *recurrent neural network* (RNN).

The Basis of Recurrence

To understand how recurrence might work, let's first consider the limitations of the models used thus far in the book. Ultimately, creating a model looks a little bit like [Figure 7-1](#). You provide data and labels and define a model architecture, and the model learns the rules that fit the data to the labels. Those rules then become available to you as an API that will give you back predicted labels for future data.

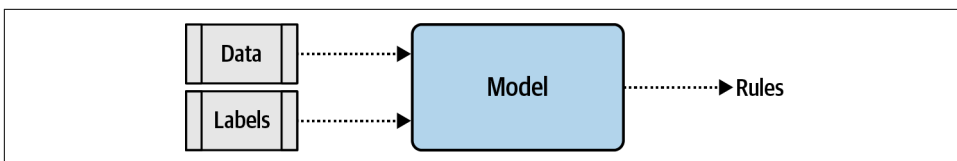


Figure 7-1. High-level view of model creation

But, as you can see, the data is lumped in wholesale. There's no granularity involved, and no effort to understand the sequence in which that data occurs. This means the words “blue” and “sky” have no different meaning in sentences such as “today I am blue, because the sky is gray” and “today I am happy, and there's a beautiful blue sky.” To us the difference in the use of these words is obvious, but to a model, with the architecture shown here, there really is no difference.

So how do we fix this? Let's first explore the nature of recurrence, and from there you'll be able to see how a basic RNN can work.

Consider the famous Fibonacci sequence of numbers. In case you aren't familiar with it, I've put some of them into [Figure 7-2](#).

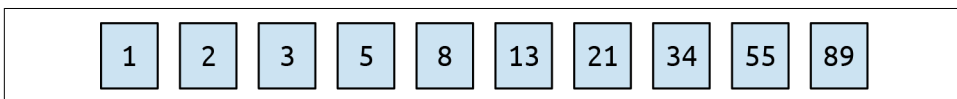


Figure 7-2. The first few numbers in the Fibonacci sequence

The idea behind this sequence is that every number is the sum of the two numbers preceding it. So if we start with 1 and 2, the next number is $1 + 2$, which is 3. The one after that is $2 + 3$, which is 5, then $3 + 5$, which is 8, and so on.

We can place this in a computational graph to get [Figure 7-3](#).

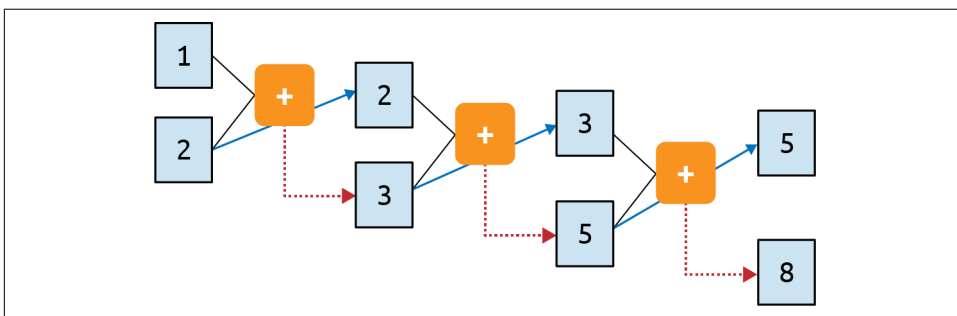


Figure 7-3. A computational graph representation of the Fibonacci sequence

Here you can see that we feed 1 and 2 into the function and get 3 as the output. We carry the second parameter (2) over to the next step, and feed it into the function

along with the output from the previous step (3). The output of this is 5, and it gets fed into the function with the second parameter from the previous step (3) to produce an output of 8. This process continues indefinitely, with every operation depending on those before it. The 1 at the top left sort of “survives” through the process. It’s an element of the 3 that gets fed into the second operation, it’s an element of the 5 that gets fed into the third, and so on. Thus, some of the essence of the 1 is preserved throughout the sequence, though its impact on the overall value is diminished.

This is analogous to how a recurrent neuron is architected. You can see the typical representation of a recurrent neuron in [Figure 7-4](#).

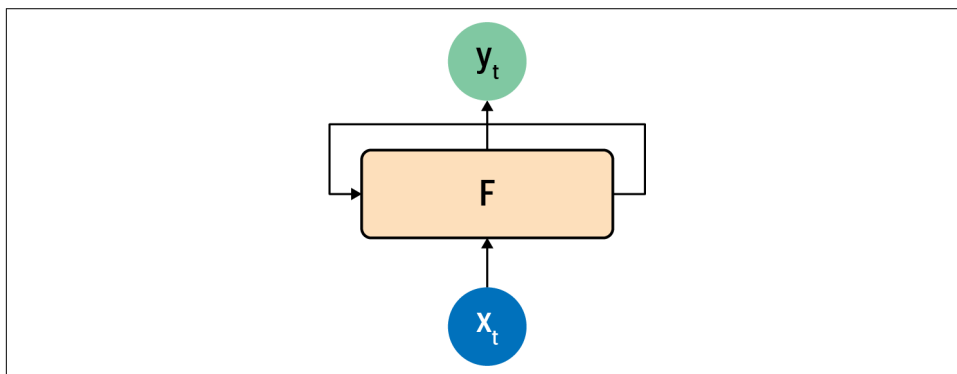


Figure 7-4. A recurrent neuron

A value x is fed into the function F at a time step, so it’s typically labeled x_t . This produces an output y at that time step, which is typically labeled y_t . It also produces a value that is fed forward to the next step, which is indicated by the arrow from F to itself.

This is made a little clearer if you look at how recurrent neurons work beside each other across time steps, which you can see in [Figure 7-5](#).

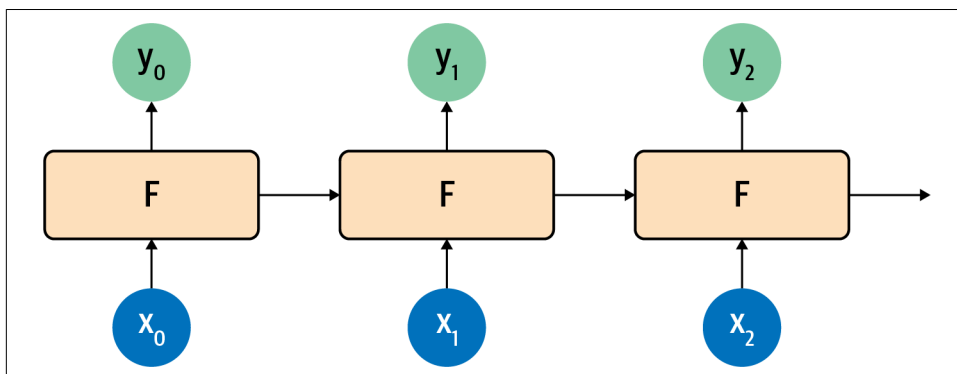


Figure 7-5. Recurrent neurons in time steps

Here, x_0 is operated on to get y_0 and a value that's passed forward. The next step gets that value and x_1 and produces y_1 and a value that's passed forward. The next one gets that value and x_2 and produces y_2 and a pass-forward value, and so on down the line. This is similar to what we saw with the Fibonacci sequence, and I always find that to be a handy mnemonic when trying to remember how an RNN works.

Extending Recurrence for Language

In the previous section you saw how a recurrent neural network operating over several time steps can help maintain context across a sequence. Indeed, RNNs will be used for sequence modeling later in this book. But there's a nuance when it comes to language that can be missed when using a simple RNN like those in [Figure 7-4](#) and [Figure 7-5](#). As in the Fibonacci sequence example mentioned earlier, the amount of context that's carried over will diminish over time. The effect of the output of the neuron at step 1 is huge at step 2, smaller at step 3, smaller still at step 4, and so on. So if we have a sentence like “Today has a beautiful blue <something>,” the word “blue” will have a strong impact on the next word; we can guess that it's likely to be “sky.” But what about context that comes from further back in a sentence? For example, consider the sentence “I lived in Ireland, so in high school I had to learn how to speak and write <something>.”

That <something> is Gaelic, but the word that really gives us that context is “Ireland,” which is much further back in the sentence. Thus, for us to be able to recognize what <something> should be, a way for context to be preserved across a longer distance is needed. The short-term memory of an RNN needs to get longer, and in recognition of this an enhancement to the architecture, called *long short-term memory* (LSTM), was invented.

While I won't go into detail on the underlying architecture of how LSTMs work, the high-level diagram shown in [Figure 7-6](#) gets the main point across. To learn more about the internal operations, check out Christopher Olah's excellent [blog post](#) on the subject.

The LSTM architecture enhances the basic RNN by adding a “cell state” that enables context to be maintained not just from step to step, but across the entire sequence of steps. Remembering that these are neurons, learning in the way neurons do, you can see that this ensures that the context that is important will be learned over time.

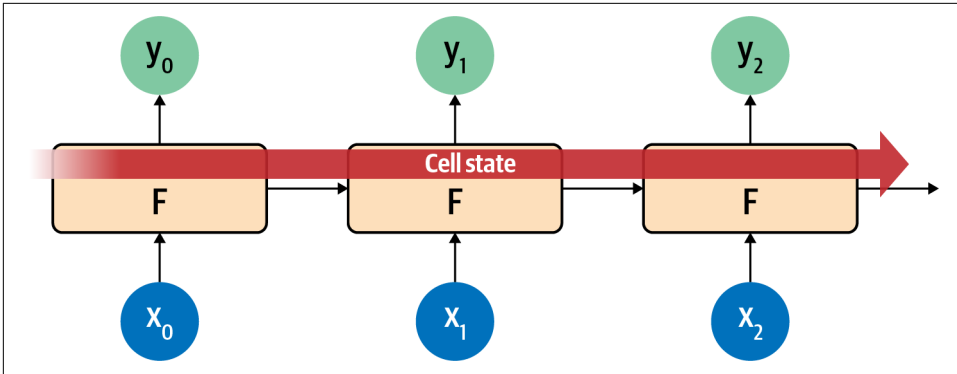


Figure 7-6. High-level view of LSTM architecture

An important part of an LSTM is that it can be bidirectional—the time steps are iterated both forward and backward, so that context can be learned in both directions. See Figure 7-7 for a high-level view of this.

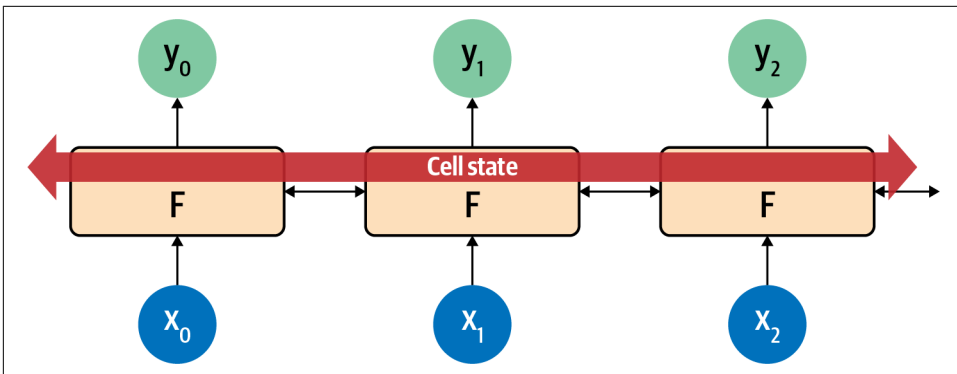


Figure 7-7. High-level view of LSTM bidirectional architecture

In this way, evaluation in the direction from 0 to *number_of_steps* is done, as is evaluation from *number_of_steps* to 0. At each step, the y result is an aggregation of the “forward” pass and the “backward” pass. You can see this in Figure 7-8.

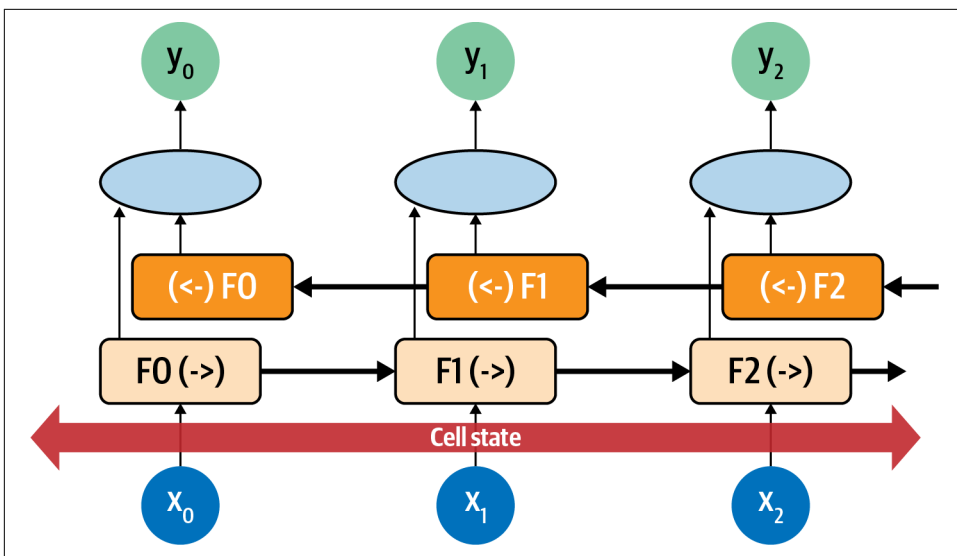


Figure 7-8. Bidirectional LSTM

Consider each neuron at each time step to be F0, F1, F2, etc. The direction of the time step is shown, so the calculation at F1 in the forward direction is F1(->), and in the reverse direction it's (->)F1. The values of these are aggregated to give the y value for that time step. Additionally, the cell state is bidirectional. This can be really useful for managing context in sentences. Again considering the sentence “I lived in Ireland, so in high school I had to learn how to speak and write <something>,” you can see how the <something> was qualified to be “Gaelic” by the context word “Ireland.” But what if it were the other way around: “I lived in <this country>, so in high school I had to learn how to speak and write Gaelic”? You can see that by going *backward* through the sentence we can learn about what <this country> should be. Thus, using bidirectional LSTMs can be very powerful for understanding sentiment in text (and as you’ll see in [Chapter 8](#), they’re really powerful for generating text too!).

Of course, there’s a lot going on with LSTMs, in particular bidirectional ones, so expect training to be slow. Here’s where it’s worth investing in a GPU, or at the very least using a hosted one in Google Colab if you can.

Creating a Text Classifier with RNNs

In [Chapter 6](#) you experimented with creating a classifier for the Sarcasm dataset using embeddings. In that case words were turned into vectors before being aggregated and then fed into dense layers for classification. When using an RNN layer such as an LSTM, you don’t do the aggregation and can feed the output of the embedding layer directly into the recurrent layer. When it comes to the dimensionality of the recurrent

layer, a rule of thumb you'll often see is that it's the same size as the embedding dimension. This isn't necessary, but can be a good starting point. Note that while in [Chapter 6](#) I mentioned that the embedding dimension is often the fourth root of the size of the vocabulary, when using RNNs you'll often see that that rule is ignored because it would make the size of the recurrent layer too small.

So, for example, the simple model architecture for the sarcasm classifier you developed in [Chapter 6](#) could be updated to this to use a bidirectional LSTM:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim)),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

The loss function and classifier can be set to this (note that the learning rate is 0.00001, or 1e-5):

```
adam = tf.keras.optimizers.Adam(learning_rate=0.00001,
                                beta_1=0.9, beta_2=0.999, amsgrad=False)

model.compile(loss='binary_crossentropy',
              optimizer=adam, metrics=['accuracy'])
```

When you print out the model architecture summary, you'll see something like this. Note that the vocab size is 20,000 and the embedding dimension is 64. This gives 1,280,000 parameters in the embedding layer, and the bidirectional layer will have 128 neurons (64 out, 64 back) :

Layer (type)	Output Shape	Param #
embedding_11 (Embedding)	(None, None, 64)	1280000
bidirectional_7 (Bidirection	(None, 128)	66048
dense_18 (Dense)	(None, 24)	3096
dense_19 (Dense)	(None, 1)	25
Total params: 1,349,169		
Trainable params: 1,349,169		
Non-trainable params: 0		

[Figure 7-9](#) shows the results of training with this over 30 epochs.

As you can see, the accuracy of the network on training data rapidly climbs above 90%, but the validation data plateaus at around 80%. This is similar to the figures we got earlier, but inspecting the loss chart in [Figure 7-10](#) shows that while the loss for

the validation set diverged after 15 epochs, it also flattened out to a much lower value than the loss charts in [Chapter 6](#), despite using 20,000 words instead of 2,000.

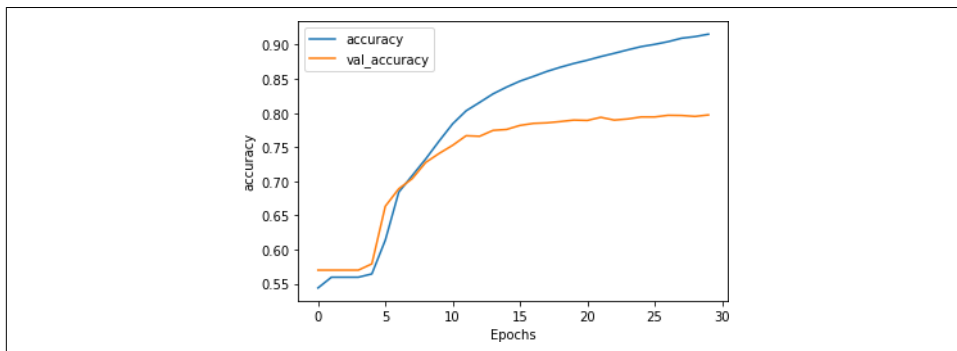


Figure 7-9. Accuracy for LSTM over 30 epochs

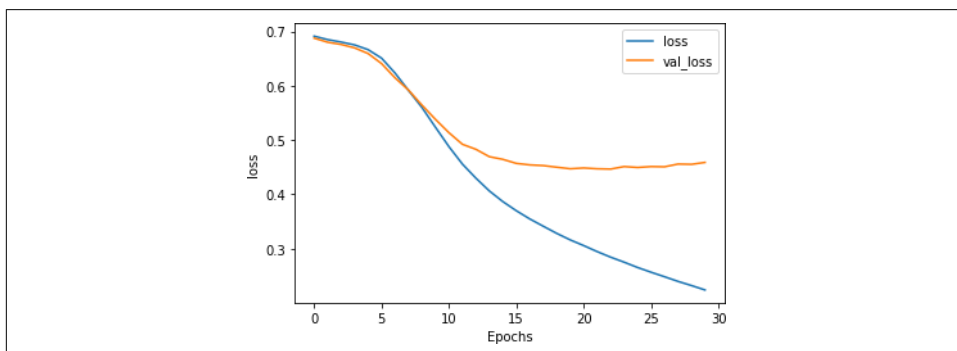


Figure 7-10. Loss with LSTM over 30 epochs

This was just using a single LSTM layer, however. In the next section you'll see how to use stacked LSTMs and explore the impact on the accuracy of classifying this dataset.

Stacking LSTMs

In the previous section you saw how to use an LSTM layer after the embedding layer to help with classifying the contents of the Sarcasm dataset. But LSTMs can be stacked on top of each other, and this approach is used in many state-of-the-art NLP models.

Stacking LSTMs with TensorFlow is pretty straightforward. You add them as extra layers just like you would with a Dense layer, but with the exception that all of the layers prior to the last one will need to have their `return_sequences` property set to `True`. Here's an example:

```

model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim,
        return_sequences=True)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim)),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

```

The final layer can also have `return_sequences=True` set, in which case it will return sequences of values to the dense layers for classification instead of single ones. This can be handy when parsing the output of the model, as we'll discuss later. The model architecture will look like this:

Layer (type)	Output Shape	Param #
embedding_12 (Embedding)	(None, None, 64)	1280000
bidirectional_8 (Bidirection	(None, None, 128)	66048
bidirectional_9 (Bidirection	(None, 128)	98816
dense_20 (Dense)	(None, 24)	3096
dense_21 (Dense)	(None, 1)	25
Total params: 1,447,985		
Trainable params: 1,447,985		
Non-trainable params: 0		

Adding the extra layer will give us roughly 100,000 extra parameters that need to be learned, an increase of about 8%. So, it might slow the network down, but the cost is relatively low if there's a reasonable benefit.

After training for 30 epochs, the result looks like [Figure 7-11](#). While the accuracy on the validation set is flat, examining the loss ([Figure 7-12](#)) tells a different story.

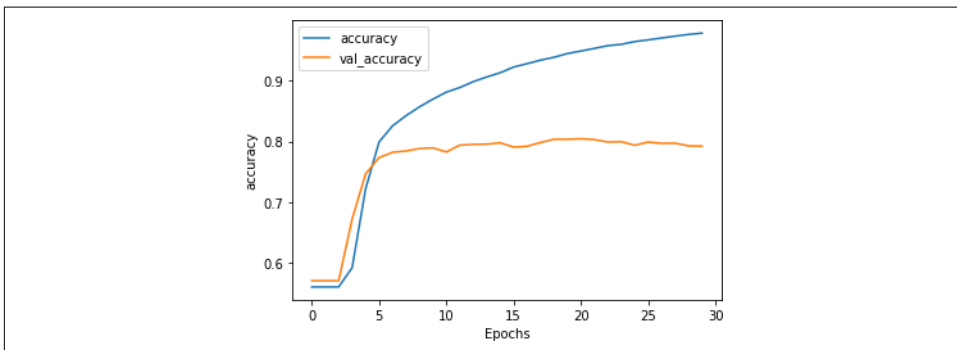


Figure 7-11. Accuracy for stacked LSTM architecture

As you can see in [Figure 7-12](#), while the accuracy for both training and validation looked good, the validation loss quickly took off upwards, a clear sign of overfitting.

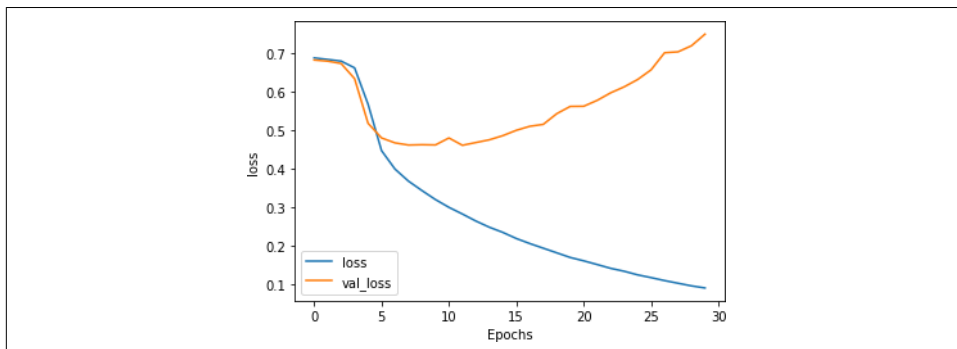


Figure 7-12. Loss for stacked LSTM architecture

This overfitting (indicated by the training accuracy climbing toward 100% as the loss falls smoothly, while the validation accuracy is relatively steady and the loss increases drastically) is a result of the model getting overspecialized for the training set. As with the examples in [Chapter 6](#), this shows that it's easy to be lulled into a false sense of security if you just look at the accuracy metrics without examining the loss.

Optimizing stacked LSTMs

In [Chapter 6](#) you saw that a very effective method to reduce overfitting was to reduce the learning rate. It's worth exploring whether that will have a positive effect on a recurrent neural network too.

For example, the following code reduces the learning rate by 20% from 0.00001 to 0.000008:

```
adam = tf.keras.optimizers.Adam(learning_rate=0.000008,  
    beta_1=0.9, beta_2=0.999, amsgrad=False)  
  
model.compile(loss='binary_crossentropy',  
    optimizer=adam,metrics=['accuracy'])
```

[Figure 7-13](#) demonstrates the impact of this on training. There doesn't seem to be much of a difference, although the curves (particularly for the validation set) are a little smoother.

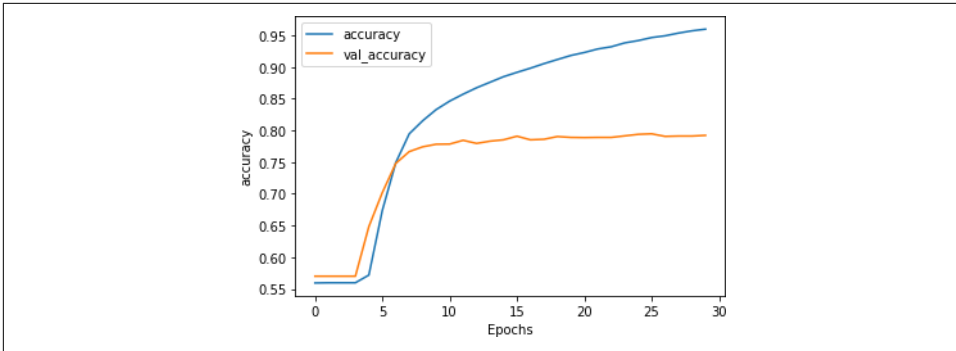


Figure 7-13. Impact of reduced learning rate on accuracy with stacked LSTMs

While an initial look at [Figure 7-14](#) similarly suggests minimal impact on loss due to the reduced learning rate, it's worth looking a little closer. Despite the shape of the curve being roughly similar, the rate of loss increase is clearly lower: after 30 epochs it's at around 0.6, whereas with the higher learning rate it was close to 0.8. Adjusting the learning rate hyperparameter certainly seems worth investigation.

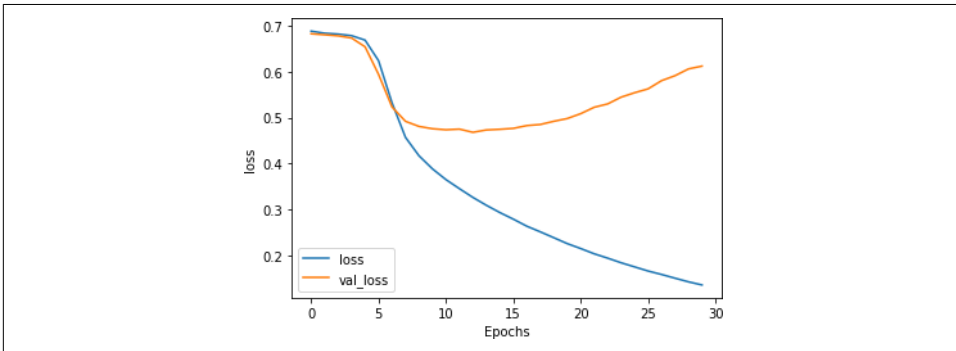


Figure 7-14. Impact of reduced learning rate on loss with stacked LSTMs

Using dropout

In addition to changing the learning rate parameter, it's also worth considering using dropout in the LSTM layers. It works exactly the same as for dense layers, as discussed in [Chapter 3](#), where random neurons are dropped to prevent a proximity bias from impacting the learning.

Dropout can be implemented using a parameter on the LSTM layer. Here's an example:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim,
        return_sequences=True, dropout=0.2)),
```



```

tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim,
    dropout=0.2)),
tf.keras.layers.Dense(24, activation='relu'),
tf.keras.layers.Dense(1, activation='sigmoid')
])

```

Do note that implementing dropout will greatly slow down your training. In my case, using Colab, it went from ~10 seconds per epoch to ~180 seconds.

The accuracy results can be seen in [Figure 7-15](#).

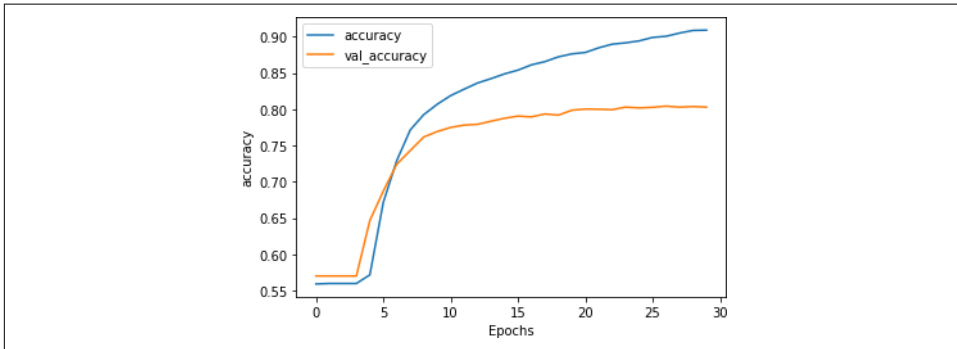


Figure 7-15. Accuracy of stacked LSTMs using dropout

As you can see, using dropout doesn't have much impact on the accuracy of the network, which is good! There's always a worry that losing neurons will make your model perform worse, but as we can see here that's not the case.

There's also a positive impact on loss, as you can see in [Figure 7-16](#).

While the curves are clearly diverging, they are closer than they were previously, and the validation set is flattening out at a loss of about 0.5. That's significantly better than the 0.8 seen previously. As this example shows, dropout is another handy technique that you can use to improve the performance of LSTM-based RNNs.

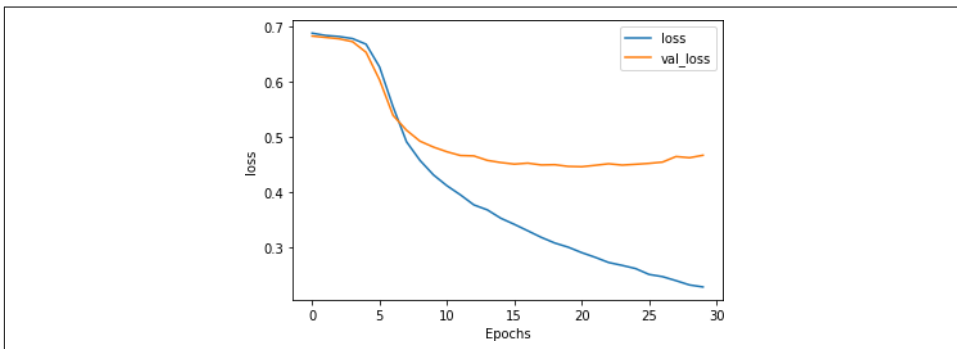


Figure 7-16. Loss curves for dropout-enabled LSTMs

It's worth exploring these techniques to avoid overfitting in your data, as well as the techniques to preprocess your data that we covered in [Chapter 6](#). But there's one thing that we haven't yet tried—a form of transfer learning where you can use pre-learned embeddings for words instead of trying to learn your own. We'll explore that next.

Using Pretrained Embeddings with RNNs

In all the previous examples, you gathered the full set of words to be used in the training set, then trained embeddings with them. These were initially aggregated before being fed into a dense network, and in this chapter you explored how to improve the results using an RNN. While doing this, you were restricted to the words in your dataset and how their embeddings could be learned using the labels from that dataset.

Think back to [Chapter 4](#), where we discussed transfer learning. What if, instead of learning the embeddings for yourself, you could instead use prelearned embeddings, where researchers have already done the hard work of turning words into vectors and those vectors are proven? One example of this is the [GloVe \(Global Vectors for Word Representation\) model](#) developed by Jeffrey Pennington, Richard Socher, and Christopher Manning at Stanford.

In this case, the researchers have shared their pretrained word vectors for a variety of datasets:

- A 6-billion-token, 400,000-word vocab set in 50, 100, 200, and 300 dimensions with words taken from Wikipedia and Gigaword
- A 42-billion-token, 1.9-million-word vocab in 300 dimensions from a common crawl
- An 840-billion-token, 2.2-million-word vocab in 300 dimensions from a common crawl
- A 27-billion-token, 1.2-million-word vocab in 25, 50, 100 and 200 dimensions from a Twitter crawl of 2 billion tweets

Given that the vectors are already pretrained, it's simple to reuse them in your TensorFlow code, instead of learning from scratch. First, you'll have to download the GloVe data. I've opted to use the Twitter data with 27 billion tokens and a 1.2-million-word vocab. The download is an archive with 25, 50, 100, and 200 dimensions.

To make it a little easier for you I've hosted the 25-dimension version, and you can download it into a Colab notebook like this:

```
!wget --no-check-certificate \
  https://storage.googleapis.com/laurencemoroney-blog.appspot.com
```

```

    /glove.twitter.27B.25d.zip \
-O /tmp/glove.zip

```

It's a ZIP file, so you can extract it like this to get a file called *glove.twitter.27b.25d.txt*:

```

# Unzip GloVe embeddings
import os
import zipfile

local_zip = '/tmp/glove.zip'
zip_ref = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall('/tmp/glove')
zip_ref.close()

```

Each entry in the file is a word, followed by the dimensional coefficients that were learned for it. The easiest way to use this is to create a dictionary where the key is the word, and the values are the embeddings. You can set up this dictionary like this:

```

glove_embeddings = dict()
f = open('/tmp/glove/glove.twitter.27B.25d.txt')
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    glove_embeddings[word] = coefs
f.close()

```

At this point you'll be able to look up the set of coefficients for any word simply by using it as the key. So, for example, to see the embeddings for “frog,” you could use:

```

glove_embeddings['frog']

```

With this resource in hand, you can use the tokenizer to get the word index for your corpus as before—but now you can create a new matrix, which I'll call the *embedding matrix*. This will use the GloVe set embeddings (taken from `glove_embeddings`) as its values. So, if you examine the words in the word index for your dataset, like this:

```

{'<OOV>': 1, 'new': 2, ... 'not': 5, 'just': 6, 'will': 7

```

Then the first row in the embedding matrix should be the coefficients from GloVe for “<OOV>,” the next row will be the coefficients for “new,” and so on.

You can create that matrix with this code:

```

embedding_matrix = np.zeros((vocab_size, embedding_dim))
for word, index in tokenizer.word_index.items():
    if index > vocab_size - 1:
        break
    else:
        embedding_vector = glove_embeddings.get(word)
        if embedding_vector is not None:
            embedding_matrix[index] = embedding_vector

```

This simply creates a matrix with the dimensions of your desired vocab size and the embedding dimension. Then, for every item in the tokenizer’s word index, you look up the coefficients from GloVe in `glove_embeddings`, and add those values to the matrix.

You then amend the embedding layer to use the pretrained embeddings by setting the `weights` parameter, and specify that you don’t want the layer to be trained by setting `trainable=False`:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim,
                              weights=[embedding_matrix], trainable=False),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim,
                                                          return_sequences=True)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim)),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

You can now train as before. However, you’ll want to consider your vocab size. One of the optimizations you did in the previous chapter to avoid overfitting was intended to prevent the embeddings becoming overburdened with learning low-frequency words; you avoided overfitting by using a smaller vocabulary of frequently used words. In this case, as the word embeddings have already been learned for you with GloVe, you could expand the vocabulary—but by how much?

The first thing to explore is how many of the words in your corpus are actually in the GloVe set. It has 1.2 million words, but there’s no guarantee it has *all* of your words.

So, here’s some code to perform a quick comparison so you can explore how large your vocab should be.

First let’s sort the data out. Create a list of Xs and Ys, where X is the word index, and Y=1 if the word is in the embeddings and 0 if it isn’t. Additionally, you can create a cumulative set, where you count the quotient of words at every time step. For example, the word “OOV” at index 0 isn’t in GloVe, so its cumulative Y would be 0. The word “new,” at the next index, is in GloVe, so it’s cumulative Y would be 0.5 (i.e., half of the words seen so far are in GloVe), and you’d continue to count that way for the entire dataset:

```
xs=[]
ys=[]
cumulative_x=[]
cumulative_y=[]
total_y=0
for word, index in tokenizer.word_index.items():
    xs.append(index)
    cumulative_x.append(index)
    if glove_embeddings.get(word) is not None:
```

```

total_y = total_y + 1
ys.append(1)
else:
    ys.append(0)
cumulative_y.append(total_y / index)

```

You then plot the Xs against the Ys with this code:

```

import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(12,2))
ax.spines['top'].set_visible(False)

plt.margins(x=0, y=None, tight=True)
#plt.axis([13000, 14000, 0, 1])
plt.fill(ys)

```

This will give you a word frequency chart, which will look something like [Figure 7-17](#).

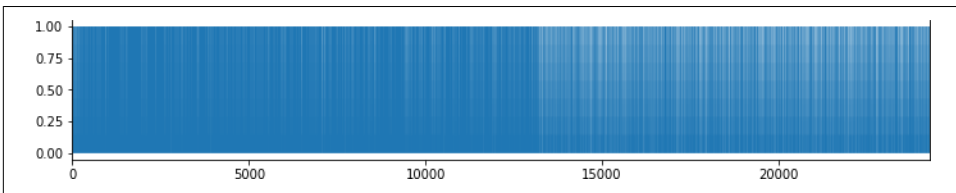


Figure 7-17. Word frequency chart

As you can see in the chart, the density changes somewhere between 10,000 and 15,000. This gives you an eyeball check that somewhere around token 13,000 the frequency of words that are *not* in the GloVe embeddings starts to outpace those that *are*.

If you then plot the `cumulative_x` against the `cumulative_y`, you can get a better sense of this. Here's the code:

```

import matplotlib.pyplot as plt
plt.plot(cumulative_x, cumulative_y)
plt.axis([0, 25000, .915, .985])

```

You can see the results in [Figure 7-18](#).

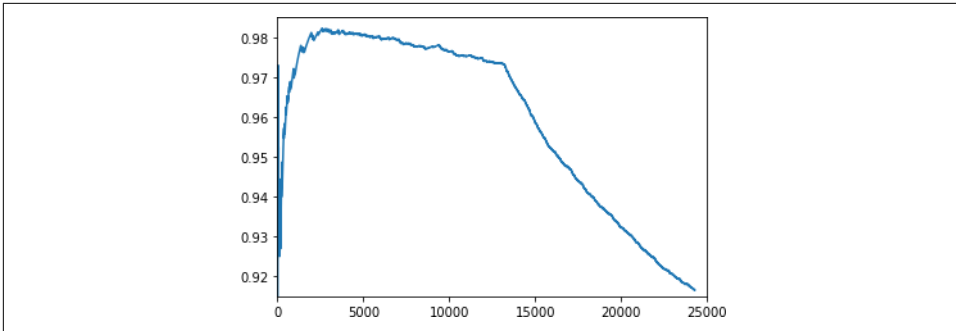


Figure 7-18. Plotting frequency of word index against GloVe

You can now tweak the parameters in `plt.axis` to zoom in to find the inflection point where words not present in GloVe begin to outpace those that are in GloVe. This is a good starting point for you to set the size of your vocabulary.

Using this method, I chose to use a vocab size of 13,200 (instead of the 2,000 that was previously used to avoid overfitting) and this model architecture, where the `embedding_dim` is 25 because of the GloVe set I'm using:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim,
        weights=[embedding_matrix], trainable=False),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim,
        return_sequences=True)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim)),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
adam = tf.keras.optimizers.Adam(learning_rate=0.00001, beta_1=0.9, beta_2=0.999,
    amsgrad=False)
model.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy'])
```

Training this for 30 epochs yields some excellent results. The accuracy is shown in [Figure 7-19](#). The validation accuracy is very close to the training accuracy, indicating that we are no longer overfitting.

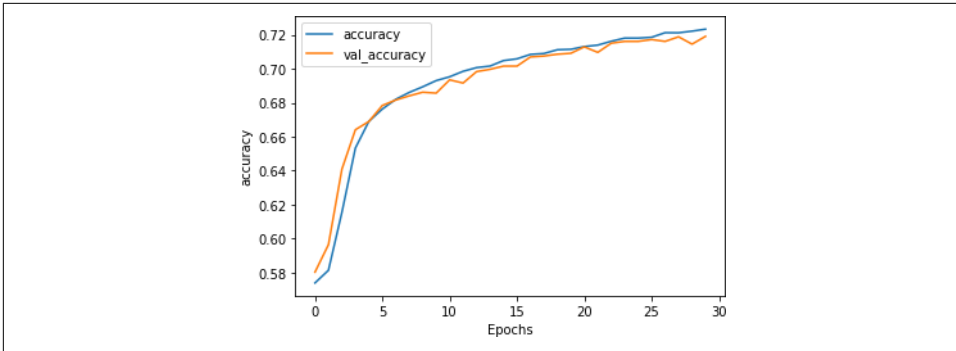


Figure 7-19. Stacked LSTM accuracy using GloVe embeddings

This is reinforced by the loss curves, as shown in Figure 7-20. The validation loss no longer diverges, showing that although our accuracy is only ~73% we can be confident that the model is accurate to that degree.

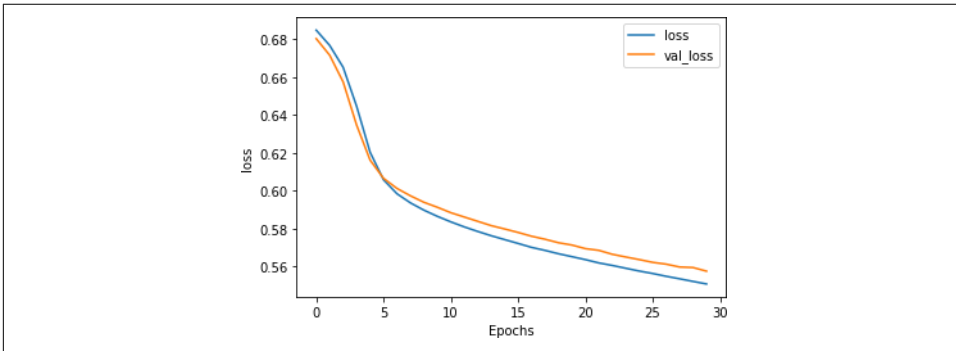


Figure 7-20. Stacked LSTM loss using GloVe embeddings

Training the model for longer shows very similar results, and indicates that while overfitting begins to occur right around epoch 80, the model is still very stable.

The accuracy metrics (Figure 7-21) show a well-trained model.

The loss metrics (Figure 7-22) show the beginning of divergence at around epoch 80, but the model still fits well.

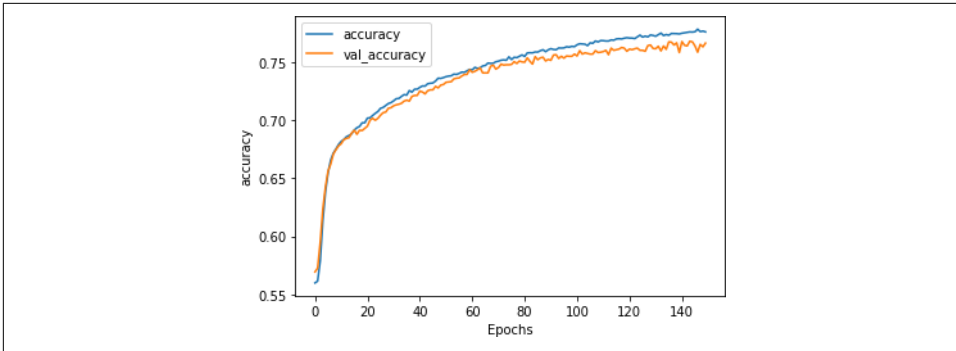


Figure 7-21. Accuracy on stacked LSTM with GloVe over 150 epochs

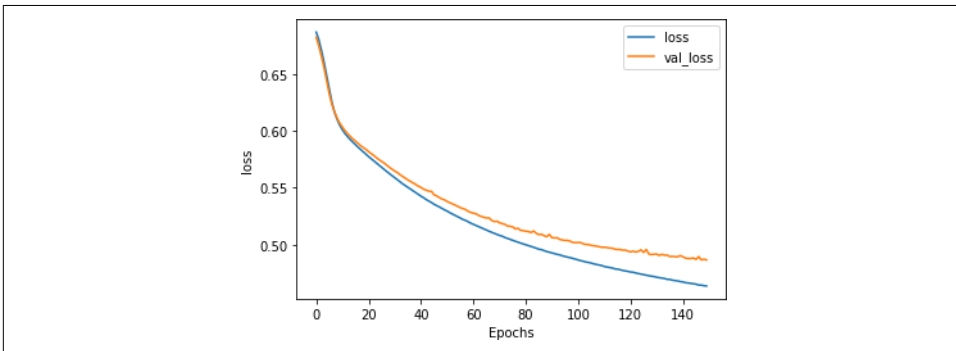


Figure 7-22. Loss on stacked LSTM with GloVe over 150 epochs

This tells us that this model is a good candidate for early stopping, where you can just train it for 75–80 epochs to get the optimal results.

I tested it with headlines from *The Onion*, source of the sarcastic headlines in the Sarcasm dataset, against other sentences, as shown here:

```
test_sentences = ["It Was, For, Uh, Medical Reasons, Says Doctor To Boris Johnson,
Explaining Why They Had To Give Him Haircut",
```

```
"It's a beautiful sunny day",
```

```
"I lived in Ireland, so in high school they made me learn to speak and write in
Gaelic",
```

```
"Census Foot Soldiers Swarm Neighborhoods, Kick Down Doors To Tally Household
Sizes"]
```


The results for these headlines are as follows—remember that values close to 50% (0.5) are considered neutral, close to 0 nonsarcastic, and close to 1 sarcastic:

```
[[0.8170955 ]  
 [0.08711044]  
 [0.61809343]  
 [0.8015281 ]]
```

The first and fourth sentences, taken from *The Onion*, showed 80%+ likelihood of sarcasm. The statement about the weather was strongly nonsarcastic (9%), and the sentence about going to high school in Ireland was deemed to be potentially sarcastic, but not with high confidence (62%).

Summary

This chapter introduced you to recurrent neural networks, which use sequence-oriented logic in their design and can help you understand the sentiment in sentences based not only on the words they contain, but also the order in which they appear. You saw how a basic RNN works, as well as how an LSTM can build on this to enable context to be preserved over the long-term. You used these to improve the sentiment analysis model you've been working on. You then looked into overfitting issues with RNNs and techniques to improve them, including using transfer learning from pre-trained embeddings. In [Chapter 8](#) you'll use what you've learned to explore how to predict words, and from there you'll be able to create a model that creates text, writing poetry for you!

Using TensorFlow to Create Text

*You know nothing, Jon Snow
the place where he's stationed
be it Cork or in the blue bird's son
sailed out to summer
old sweet long and gladness rings
so i'll wait for the wild colleen dying*

This text was generated by a very simple model trained on a small corpus. I've enhanced it a little by adding line breaks and punctuation, but other than the first line, the rest was all generated by the model you'll learn how to build in this chapter. It's kind of cool that it mentions a *wild colleen dying*—if you've watched the show that Jon Snow comes from, you'll understand why!

In the last few chapters you saw how you can use TensorFlow with text-based data, first tokenizing it into numbers and sequences that can be processed by a neural network, then using embeddings to simulate sentiment using vectors, and finally using deep and recurrent neural networks to classify text. We used the Sarcasm dataset, a small and simple one, to illustrate how all this works. In this chapter we're going to switch gears: instead of classifying existing text, you'll create a neural network that can *predict* text. Given a corpus of text, it will attempt to understand the patterns of words within it so that it can, given a new piece of text called a *seed*, predict what word should come next. Once it has that, the seed and the predicted word become the new seed, and the next word can be predicted. Thus, when trained on a corpus of text, a neural network can attempt to write new text in a similar style. To create the piece of poetry above, I collected lyrics from a number of traditional Irish songs, trained a neural network with them, and used it to predict words.

We'll start simple, using a small amount of text to illustrate how to build up to a predictive model, and we'll end by creating a full model with a lot more text. After that you can try it out to see what kind of poetry it can create!

To get started, you'll have to treat the text a little differently from what you've been doing thus far. In the previous chapters, you took sentences and turned them into sequences that were then classified based on the embeddings for the tokens within them.

When it comes to creating data that can be used for training a predictive model like this one, there's an additional step where the sequences need to be transformed into *input sequences* and *labels*, where the input sequence is a group of words and the label is the next word in the sentence. You can then train a model to match the input sequences to their labels, so that future predictions can pick a label that's close to the input sequence.

Turning Sequences into Input Sequences

When predicting text, you need to train a neural network with an input sequence (feature) that has an associated label. Matching sequences to labels is the key to predicting text.

So, for example, if in your corpus you have the sentence “Today has a beautiful blue sky,” you could split this into “Today has a beautiful blue” as the feature and “sky” as the label. Then, if you were to get a prediction for the text “Today has a beautiful blue,” it would likely be “sky.” If in the training data you also have “Yesterday had a beautiful blue sky,” split in the same way, and you were to get a prediction for the text “Tomorrow will have a beautiful blue,” then there's a high probability that the next word will be “sky.”

Given lots of sentences, training on sequences of words with the next word being the label, you can quickly build up a predictive model where the most likely next word in the sentence can be predicted from an existing body of text.

We'll start with a very small corpus of text—an excerpt from a traditional Irish song from the 1860s, some of the lyrics of which are as follows:

*In the town of Athy one Jeremy Lanigan
Battered away til he hadnt a pound.
His father died and made him a man again
Left him a farm and ten acres of ground.*

*He gave a grand party for friends and relations
Who didnt forget him when come to the wall,*

*And if youll but listen Ill make your eyes glisten
Of the rows and the ructions of Lanigan's Ball.*

*Myself to be sure got free invitation,
For all the nice girls and boys I might ask,
And just in a minute both friends and relations
Were dancing round merry as bees round a cask.*

*Judy ODaly, that nice little milliner,
She tipped me a wink for to give her a call,
And I soon arrived with Peggy McGilligan
Just in time for Lanigans Ball.*

Create a single string with all the text, and set that to be your data. Use `\n` for the line breaks. Then this corpus can be easily loaded and tokenized like this:

```
tokenizer = Tokenizer()

data="In the town of Athy one Jeremy Lanigan \n Battered away ... ..."
corpus = data.lower().split("\n")

tokenizer.fit_on_texts(corpus)
total_words = len(tokenizer.word_index) + 1
```

The result of this process is to replace the words by their token values, as shown in [Figure 8-1](#).

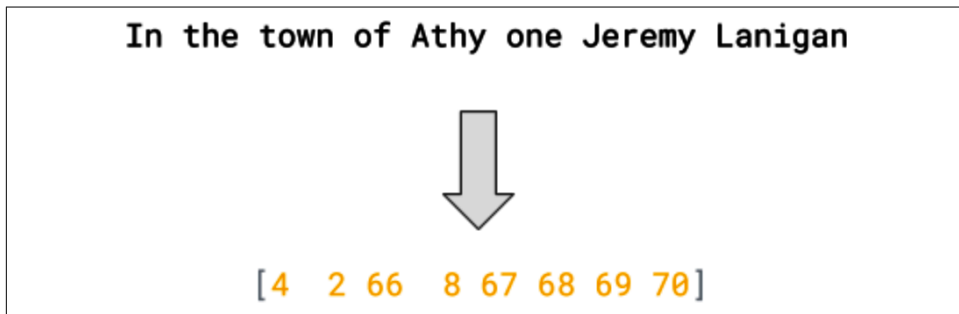


Figure 8-1. Tokenizing a sentence

To train a predictive model, we should take a further step here—splitting the sentence into multiple smaller sequences, so, for example, we can have one sequence consisting of the first two tokens, another of the first three, etc. ([Figure 8-2](#)).

Line:	Input sequences:
[4 2 66 8 67 68 69 70]	[4 2]
	[4 2 66]
	[4 2 66 8]
	[4 2 66 8 67]
	[4 2 66 8 67 68]
	[4 2 66 8 67 68 69]
	[4 2 66 8 67 68 69 70]

Figure 8-2. Turning a sequence into a number of input sequences

To do this you'll need to go through each line in the corpus and turn it into a list of tokens using `texts_to_sequences`. Then you can split each list by looping through each token and making a list of all the tokens up to it.

Here's the code:

```
input_sequences = []
for line in corpus:
    token_list = tokenizer.texts_to_sequences([line])[0]
    for i in range(1, len(token_list)):
        n_gram_sequence = token_list[:i+1]
        input_sequences.append(n_gram_sequence)

print(input_sequences[:5])
```

Once you have these input sequences, you can pad them into a regular shape. We'll use prepadding (Figure 8-3).

Line:	Padded input sequences:
[4 2 66 8 67 68 69 70]	[0 0 0 0 0 0 0 0 0 0 4 2]
	[0 0 0 0 0 0 0 0 0 0 4 2 66]
	[0 0 0 0 0 0 0 0 0 4 2 66 8]
	[0 0 0 0 0 0 0 4 2 66 8 67]
	[0 0 0 0 0 0 4 2 66 8 67 68]
	[0 0 0 0 0 4 2 66 8 67 68 69]
	[0 0 0 0 4 2 66 8 67 68 69 70]

Figure 8-3. Padding the input sequences

To do this, you'll need to find the longest sentence in the input sequences, and pad everything to that length. Here's the code:

```
max_sequence_len = max([len(x) for x in input_sequences])

input_sequences = np.array(pad_sequences(input_sequences,
                                       maxlen=max_sequence_len, padding='pre'))
```

Finally, once you have a set of padded input sequences, you can split these into features and labels, where the label is simply the last token in the input sequence (Figure 8-4).

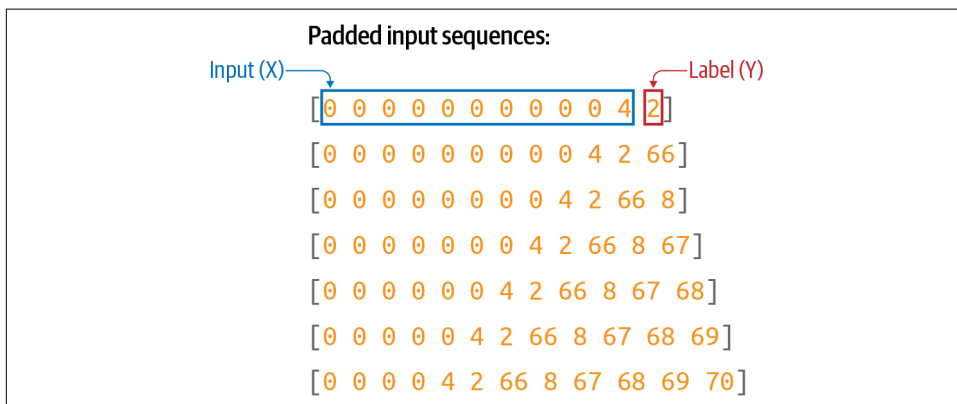


Figure 8-4. Turning the padded sequences into features (x) and labels (y)

When training a neural network, you're going to match each feature to its corresponding label. So, for example, the label for [0 0 0 0 4 2 66 8 67 68 69] will be [70].

Here's the code to separate the labels from the input sequences:

```
xs, labels = input_sequences[:, :-1], input_sequences[:, -1]
```

Next, you need to encode the labels. Right now they're just tokens—for example, the number 2 at the top of Figure 8-4. But if you want to use a token as a label in a classifier, it will have to be mapped to an output neuron. Thus, if you're going to classify n words, with each word being a class, you'll need to have n neurons. Here's where it's important to control the size of the vocabulary, because the more words you have, the more classes you'll need. Remember back in Chapters 2 and 3 when you were classifying fashion items with the Fashion MNIST dataset, and you had 10 types of items of clothing? That required you to have 10 neurons in the output layer. In this case, what if you want to predict up to 10,000 vocabulary words? You'll need an output layer with 10,000 neurons!

Additionally, you need to one-hot encode your labels so that they match the desired output from a neural network. Consider Figure 8-4. If a neural network is fed the input X consisting of a series of 0s followed by a 4, you'll want the prediction to be 2,

but how the network delivers that is by having an output layer of *vocabulary_size* neurons, where the second one has the highest probability.

To encode your labels into a set of Ys that you can then use to train, you can use the `to_categorical` utility in `tf.keras`:

```
ys = tf.keras.utils.to_categorical(labels, num_classes=total_words)
```

You can see this visually in [Figure 8-5](#).

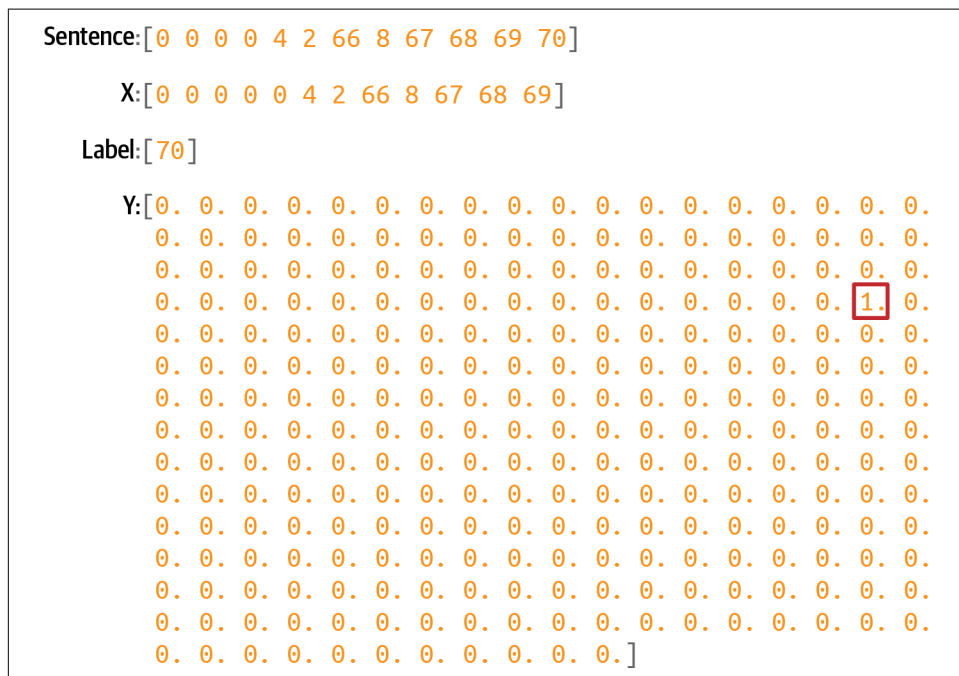


Figure 8-5. One-hot encoding labels

This is a very sparse representation, which, if you have a lot of training data and a lot of potential words, will eat memory very quickly! Suppose you had 100,000 training sentences, with a vocabulary of 10,000 words—you'd need 1,000,000,000 bytes just to hold the labels! But it's the way we have to design our network if we're going to classify and predict words.

Creating the Model

Let's now create a simple model that can be trained with this input data. It will consist of just an embedding layer, followed by an LSTM, followed by a dense layer.

For the embedding you'll need one vector per word, so the parameters will be the total number of words and the number of dimensions you want to embed on. In this case we don't have many words, so eight dimensions should be enough.

You can make the LSTM bidirectional, and the number of steps can be the length of a sequence, which is our max length minus 1 (because we took one token off the end to make the label).

Finally, the output layer will be a dense layer with the total number of words as a parameter, activated by softmax. Each neuron in this layer will be the probability that the next word matches the word for that index value:

```
model = Sequential()
model.add(Embedding(total_words, 8))
model.add(Bidirectional(LSTM(max_sequence_len-1)))
model.add(Dense(total_words, activation='softmax'))
```

Compile the model with a categorical loss function such as categorical cross entropy and an optimizer like Adam. You can also specify that you want to capture metrics:

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam', metrics=['accuracy'])
```

It's a very simple model without a lot of data, so you can train for a long time—say, 1,500 epochs:

```
history = model.fit(xs, ys, epochs=1500, verbose=1)
```

After 1,500 epochs, you'll see that it has reached very high accuracy ([Figure 8-6](#)).

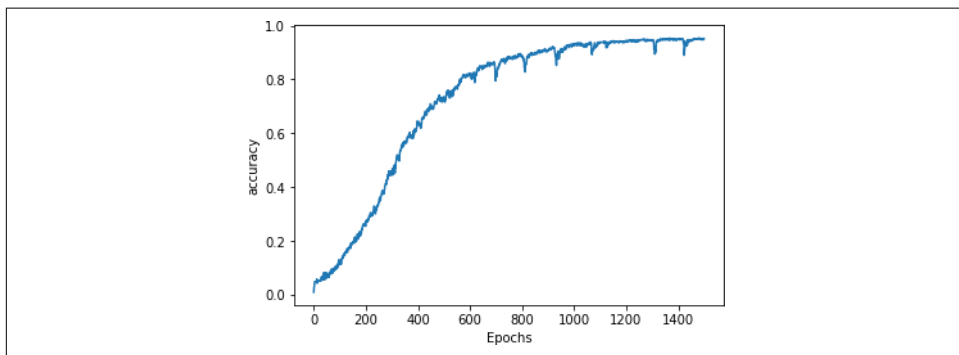


Figure 8-6. Training accuracy

With the model at around 95% accuracy, we can be assured that if we have a string of text that it has already seen it will predict the next word accurately about 95% of the time. Note, however, that when generating text it will continually see words that it hasn't previously seen, so despite this good number, you'll find that the network will rapidly end up producing nonsensical text. We'll explore this in the next section.

Generating Text

Now that you’ve trained a network that can predict the next word in a sequence, the next step is to give it a sequence of text and have it predict the next word. Let’s take a look at how to do that.

Predicting the Next Word

You’ll start by creating a phrase called the seed text. This is the initial expression on which the network will base all the content it generates. It will do this by predicting the next word.

Start with a phrase that the network has *already* seen, “in the town of athy”:

```
seed_text = "in the town of athy"
```

Next you need to tokenize this using `texts_to_sequences`. This returns an array, even if there’s only one value, so take the first element in that array:

```
token_list = tokenizer.texts_to_sequences([seed_text])[0]
```

Then you need to pad that sequence to get it into the same shape as the data used for training:

```
token_list = pad_sequences([token_list],  
                           maxlen=max_sequence_len-1, padding='pre')
```

Now you can predict the next word for this token list by calling `model.predict` on the token list. This will return the probabilities for each word in the corpus, so pass the results to `np.argmax` to get the most likely one:

```
predicted = np.argmax(model.predict(token_list), axis=-1)  
print(predicted)
```

This should give you the value 68. If you look at the word index, you’ll see that this is the word “one”:

```
'town': 66, 'athy': 67, 'one': 68, 'jeremy': 69, 'lanigan': 70,
```

You can look it up in code by searching through the word index items until you find predicted and printing it out:

```
for word, index in tokenizer.word_index.items():  
    if index == predicted:  
        print(word)  
        break
```

So, starting from the text “in the town of athy,” the network predicted the next word should be “one”—which if you look at the training data is correct, because the song begins with the line:

*In the town of Athy one Jeremy Lanigan
Battered away til he hadnt a pound*

Now that you’ve confirmed the model is working, you can get creative and use different seed text. For example, when I used the seed text “sweet jeremy saw dublin,” the next word it predicted was “then.” (This text was chosen because all of those words are in the corpus. You should expect more accurate results, at least at the beginning, for the predicted words in such cases.)

Compounding Predictions to Generate Text

In the previous section you saw how to use the model to predict the next word given a seed text. To have the neural network now create new text, you simply repeat the prediction, adding new words each time.

For example, earlier when I used the phrase “sweet jeremy saw dublin,” it predicted the next word would be “then.” You can build on this by appending “then” to the seed text to get “sweet jeremy saw dublin then” and getting another prediction. Repeating this process will give you an AI-created string of text.

Here’s the updated code from the previous section that performs this loop a number of times, with the number set by the `next_words` parameter:

```
seed_text = "sweet jeremy saw dublin"
next_words=10

for _ in range(next_words):
    token_list = tokenizer.texts_to_sequences([seed_text])[0]
    token_list = pad_sequences([token_list],
                              maxlen=max_sequence_len-1, padding='pre')
    predicted = model.predict_classes(token_list, verbose=0)
    output_word = ""

    for word, index in tokenizer.word_index.items():
        if index == predicted:
            output_word = word
            break
    seed_text += " " + output_word

print(seed_text)
```

This will end up creating a string something like this:

```
sweet jeremy saw dublin then got there as me me a call doing me
```

It rapidly descends into gibberish. Why? The first reason is that the body of training text is really small, so it has very little context to work with. The second is that the prediction of the next word in the sequence depends on the previous words in the sequence, and if there is a poor match on the previous ones, even the best “next” match will have a low probability. When you add this to the sequence and predict the next word after that, the likelihood of it having a low probability is even higher—thus, the predicted words will seem semirandom.

So, for example, while all of the words in the phrase “sweet jeremy saw dublin” exist in the corpus, they never exist in that order. When the first prediction was done, the word “then” was chosen as the most likely candidate, and it had quite a high probability (89%). When it was added to the seed to get “sweet jeremy saw dublin then,” we had another phrase not seen in the training data, so the prediction gave the highest probability to the word “got,” at 44%. Continuing to add words to the sentence reduces the likelihood of a match in the training data, and as such the prediction accuracy will suffer—leading to a more random “feel” to the words being predicted.

This leads to the phenomenon of AI-generated content getting increasingly nonsensical over time. For an example, check out the excellent sci-fi short *Sunspring*, which was written entirely by an LSTM-based network, like the one you’re building here, trained on science fiction movie scripts. The model was given seed content and tasked with generating a new script. The results were hilarious, and you’ll see that while the initial content makes sense, as the movie progresses it becomes less and less comprehensible.

Extending the Dataset

The same pattern that you used for the hardcoded dataset can be extended to use a text file very simply. I’ve hosted a text file containing about 1,700 lines of text gathered from a number of songs that you can use for experimentation. With a little modification, you can use this instead of the single hardcoded song.

To download the data in Colab, use the following code:

```
!wget --no-check-certificate \
  https://storage.googleapis.com/laurencemoroney-blog.appspot.com/ \
  irish-lyrics-eof.txt-O /tmp/irish-lyrics-eof.txt
```

Then you can simply load the text from it into your corpus like this:

```
data = open('/tmp/irish-lyrics-eof.txt').read()
corpus = data.lower().split("\n")
```

The rest of your code will then work without modification!

Training this for one thousand epochs brings you to about 60% accuracy, with the curve flattening out (Figure 8-7).

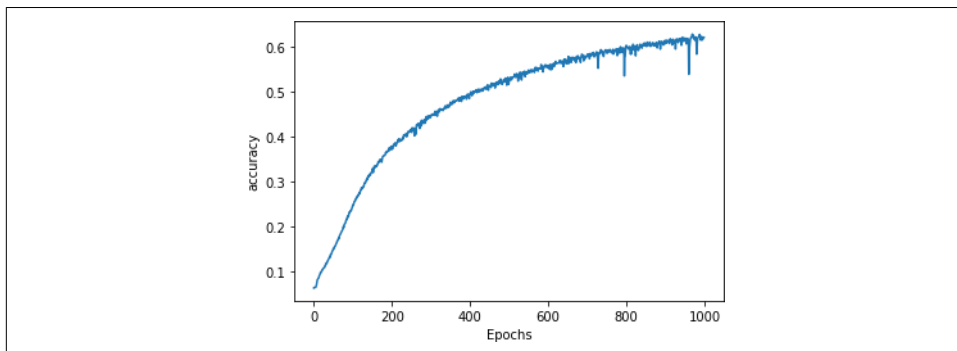


Figure 8-7. Training on a larger dataset

Trying the phrase “in the town of athy” again yields a prediction of “one,” but this time with only a 40% probability.

For “sweet jeremy saw dublin” the predicted next word is “drawn,” with a probability of 59%. Predicting the next 10 words yields:

sweet jeremy saw dublin drawn and fondly i am dead and the parting graceful

It’s looking a little better! But can we improve it further?

Changing the Model Architecture

One way that you can improve the model is to change its architecture, using multiple stacked LSTMs. This is pretty straightforward—just ensure that you set `return_sequences` to `True` on the first of them. Here’s the code:

```
model = Sequential()
model.add(Embedding(total_words, 8))
model.add(Bidirectional(LSTM(max_sequence_len-1, return_sequences='True'))))
model.add(Bidirectional(LSTM(max_sequence_len-1)))
model.add(Dense(total_words, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam',
              metrics=['accuracy'])
history = model.fit(xs, ys, epochs=1000, verbose=1)
```

You can see the impact this has on training for one thousand epochs in Figure 8-8. It’s not significantly different from the previous curve.

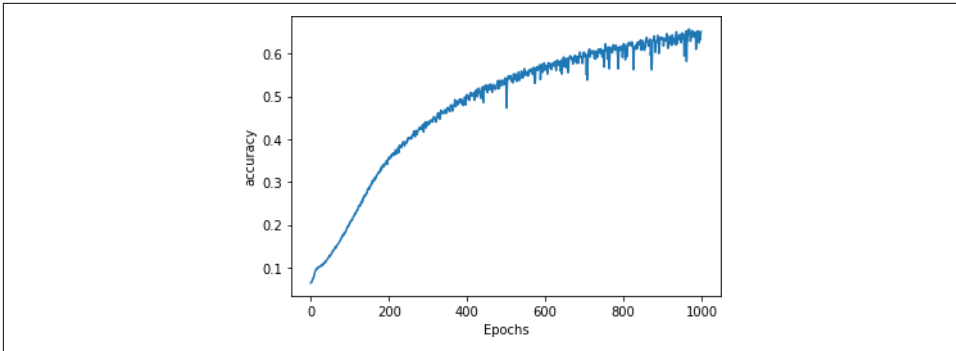


Figure 8-8. Adding a second LSTM layer

When testing with the same phrases as before, this time I got “more” as the next word after “in the town of athy” with a 51% probability, and after “sweet jeremy saw dublin” I got “cailín” (the Gaelic word for “girl”) with a 61% probability. Again, when predicting more words, the output quickly descended into gibberish.

Here are some examples:

```
sweet jeremy saw dublin cailin loo ra fountain plundering that fulfill  
you mccarthy you mccarthy down
```

```
you know nothing jon snow johnny cease and she danced that put to smother well  
i must the wind flowers  
dreams it love to laid ned the mossy and night i weirs
```

If you get different results, don’t worry—you didn’t do anything wrong, but the random initialization of the neurons will impact the final scores.

Improving the Data

There’s a small trick that you can use to extend the size of this dataset without adding any new songs, called *windowing* the data. Right now, every line in every song is read as a single line and then turned into input sequences, as you saw in [Figure 8-2](#). While humans read songs line by line in order to hear rhyme and meter, the model doesn’t have to, in particular when using bidirectional LSTMs.

So, instead of taking the line “In the town of Athy, one Jeremy Lanigan,” processing that, and then moving to the next line (“Battered away till he hadn’t a pound”) and processing that, we could treat all the lines as one long, continuous text. We can then create a “window” into that text of n words, process that, and then move the window forward one word to get the next input sequence ([Figure 8-9](#)).

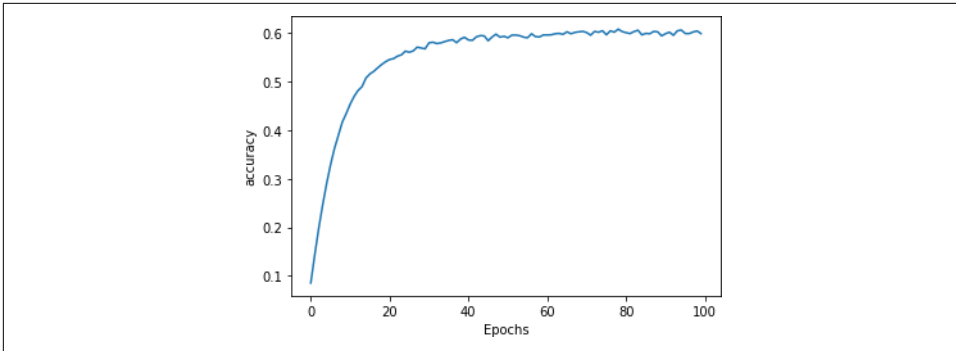


Figure 8-9. A moving word window

In this case, far more training data can be yielded in the form of an increased number of input sequences. Moving the window across the entire corpus of text would give us $((\text{number_of_words} - \text{window_size}) \times \text{window_size})$ input sequences that we could train with.

The code is pretty simple—when loading the data, instead of splitting each song line into a “sentence,” we can create them on the fly from the words in the corpus:

```

window_size=10
sentences=[]
alltext=[]
data = open('/tmp/irish-lyrics-eof.txt').read()
corpus = data.lower()
words = corpus.split(" ")
range_size = len(words)-max_sequence_len
for i in range(0, range_size):
    thissentence=""
    for word in range(0, window_size-1):
        word = words[i+word]
        thissentence = thissentence + word
        thissentence = thissentence + " "
    sentences.append(thissentence)

```

In this case, because we no longer have sentences and we’re creating sequences the same size as the moving window, `max_sequence_len` is the size of the window. The full file is read, converted to lowercase, and split into an array of words using string splitting. The code then loops through the words and makes sentences of each word from the current index up to the current index plus the window size, adding each of those newly constructed sentences to the sentences array.

When training, you’ll notice that the extra data makes it much slower per epoch, but the results are greatly improved, and the generated text descends into gibberish much more slowly.

Here's an example that caught my eye—particularly the last line!

*you know nothing, jon snow is gone
and the young and the rose and wide
to where my love i will play
the heart of the kerry
the wall i watched a neat little town*

There are many hyperparameters you can try tuning. Changing the window size will change the amount of training data—a smaller window size can yield more data, but there will be fewer words to give to a label, so if you set it too small you'll end up with nonsensical poetry. You can also change the dimensions in the embedding, the number of LSTMs, or the size of the vocab to use for training. Given that percentage accuracy isn't the best measurement—you'll want to make a more subjective examination of how much “sense” the poetry makes—there's no hard-and-fast rule to follow to determine whether your model is “good” or not.

For example, when I tried using a window size of 6, increasing the number of dimensions for the embedding to 16, changing the number of LSTMs from the window size (which would be 6) to 32, and upping the learning rate on the Adam optimizer, I got a nice, smooth, learning curve (Figure 8-10) and some of the poetry began to make more sense.

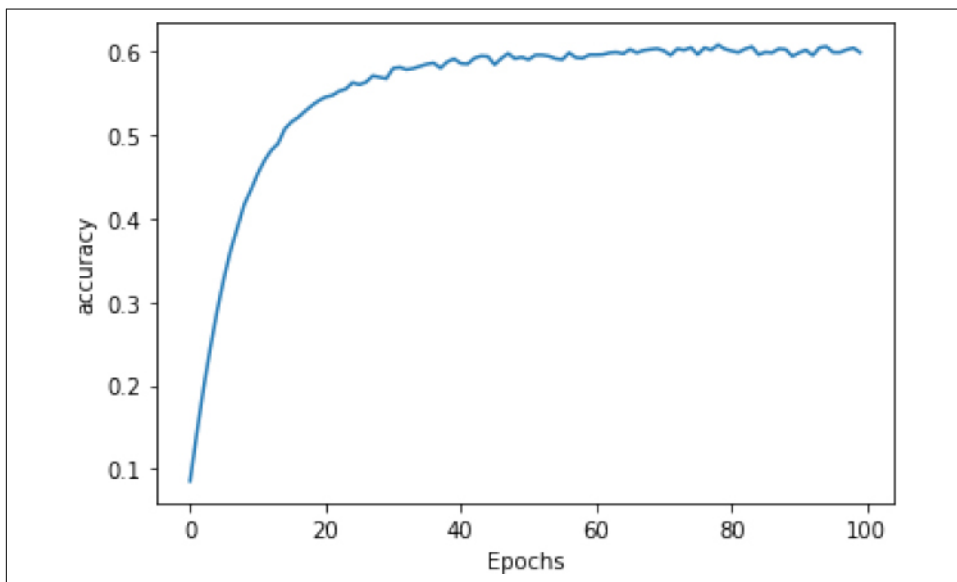


Figure 8-10. Learning curve with adjusted hyperparameters

When using “sweet jeremy saw dublin” as the seed (remember, all of the words in the seed are in the corpus), I got this poem:

*sweet jeremy saw dublin
whack fol
all the watch came
and if ever you love get up from the stool
longs to go as i was passing my aged father
if you can visit new ross
gallant words i shall make
such powr of her goods
and her gear
and her calico blouse
she began the one night
rain from the morning so early
oer railroad ties and crossings
i made my weary way
through swamps and elevations
my tired feet
was the good heavens*

While the phrase “whack fol” may not make sense to many readers, it’s commonly heard in some Irish songs, kind of like “la la la” or “doobie-doobie-doo.” What I really liked about this was how some of the *later* phrases kept some kind of sense, like “such power of her good and her gear, and her calico blouse”—but this could be due to overfitting to the phrases that already exist in the songs within the corpus. For example, the lines beginning “oer railroad ties...” through “my tired feet” are taken directly from a song called “The Lakes of Pontchartrain” which is in the corpus. If you encounter issues like this, it’s best to reduce the learning rate and maybe decrease the number of LSTMs. But above all, experiment and have fun!

Character-Based Encoding

For the last few chapters we’ve been looking at NLP using word-based encoding. I find that much easier to get started with, but when it comes to generating text, you might also want to consider character-based encoding because the number of unique *characters* in a corpus tends to be a lot less than the number of unique *words*. As such, you can have a lot fewer neurons in your output layer, and your output predictions are spread across fewer probabilities. For example, when looking at the dataset of **the complete works of Shakespeare**, you’ll see that there are only 65 unique characters in the entire set. So, when you are making predictions, instead of looking at probabilities of the next word across 2,700 words as in the Irish songs dataset, you’re only looking at 65. This makes your model a bit simpler!

What's also nice about character encoding is that punctuation characters are also included, so line breaks etc. can be predicted. As an example, when I used an RNN trained on the Shakespeare corpus to predict the text following on from my favorite *Game of Thrones* line, I got:

YGRITTE:

You know nothing, Jon Snow.

Good night, we'll prove those body's servants to

The traitor be these mine:

So diswarl his body in hope in this resceins,

I cannot judg appeal't.

MENENIUS:

Why, 'tis pompetsion.

KING RICHARD II:

I think he make her thought on mine;

She will not: suffer up thy bonds:

How doched it, I pray the gott,

We'll no fame to this your love, and you were ends

It's kind of cool that she identifies him as a traitor and wants to tie him up ("diswarl his body"), but I have no idea what "resceins" means! If you watch the show, this is part of the plot, so maybe Shakespeare was on to something without realizing it!

Of course, I do think we tend to be a little more forgiving when using something like Shakespeare's texts as our training data, because the language is already a little unfamiliar.

As with the Irish songs model, the output does quickly degenerate into nonsensical text, but it's still fun to play with. To try it for yourself, you can check out the [Colab](#).

Summary

In this chapter we explored how to do basic text generation using a trained LSTM-based model. You saw how you can split text into training features and labels, using words as labels, and create a model that, when given seed text, can predict the next likely word. You iterated on this to improve the model for better results, exploring a dataset of traditional Irish songs. You also saw a little about how this could potentially be improved with character-based text generation with an example that uses Shakespearean text. Hopefully this was a fun introduction to how machine learning models can synthesize text!

Understanding Sequence and Time Series Data

Time series are everywhere. You've probably seen them in things like weather forecasts, stock prices, and historic trends like Moore's law (Figure 9-1). If you're not familiar with Moore's law, it predicts that the number of transistors on a microchip will roughly double every two years. For almost 50 years it has proven to be an accurate predictor of the future of computing power and cost.

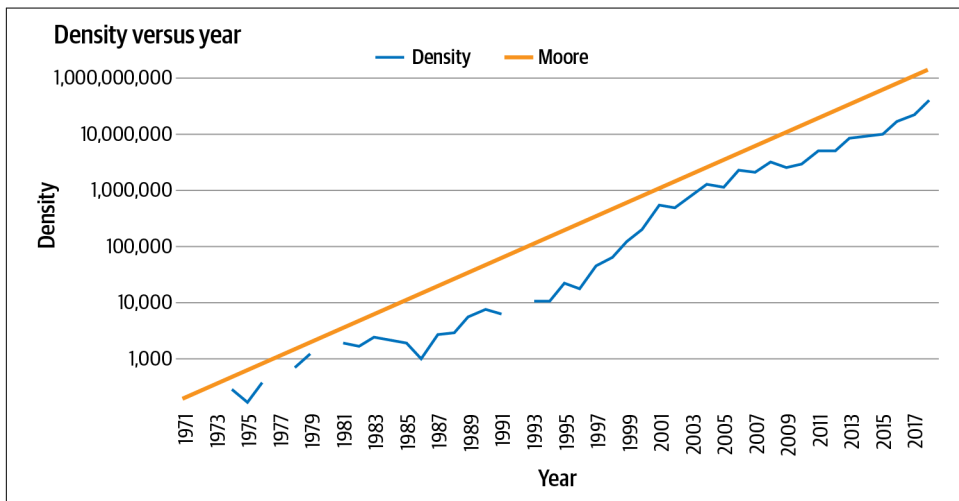


Figure 9-1. Moore's law

Time series data is a set of values that are spaced over time. When plotted, the x-axis is usually temporal in nature. Often there are a number of values plotted on the time axis, such as in this example where the number of transistors is one plot and the predicted value from Moore's law is the other. This is called a *multivariate* time series. If there's just a single value—for example, volume of rainfall over time—it's called a *univariate* time series.

With Moore's law, predictions are simple because there's a fixed and simple rule that allows us to roughly predict the future—a rule that has held for about 50 years.

But what about a time series like that in **Figure 9-2**?

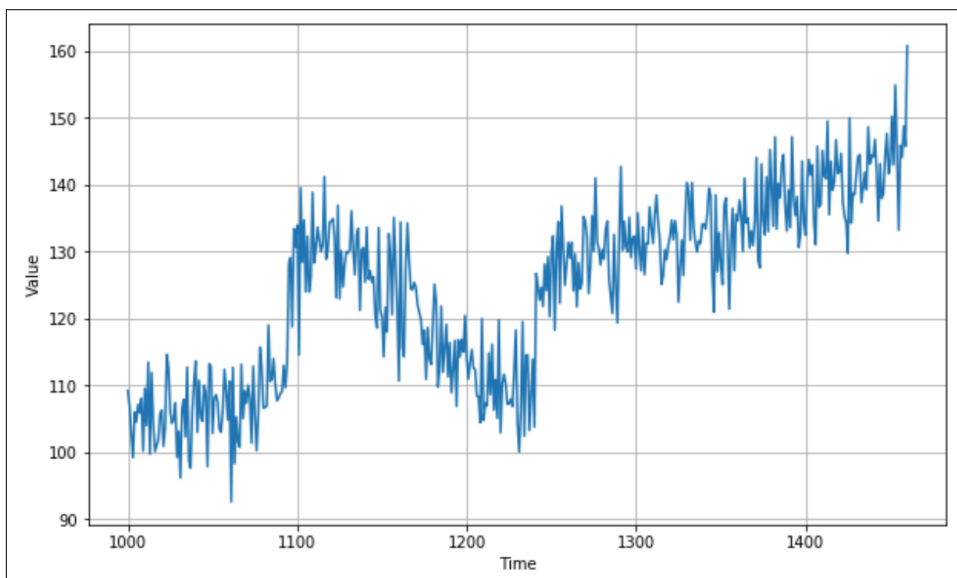


Figure 9-2. A real-world time series

While this time series was artificially created (you'll see how to do that later in this chapter), it has all the attributes of a complex real-world time series like a stock chart or seasonal rainfall. Despite the seeming randomness, time series have some common attributes that are helpful in designing ML models that can predict them, as described in the next section.

Common Attributes of Time Series

While time series might appear random and noisy, often there are common attributes that are predictable. In this section we'll explore some of these.

Trend

Time series typically move in a specific direction. In the case of Moore's law, it's easy to see that over time the values on the y-axis increase, and there's an upward trend. There's also an upward trend in the time series in [Figure 9-2](#). Of course, this won't always be the case: some time series may be roughly level over time, despite seasonal changes, and others have a downward trend. For example, this is the case in the inverse version of Moore's law that predicts the price per transistor.

Seasonality

Many time series have a repeating pattern over time, with the repeats happening at regular intervals called *seasons*. Consider, for example, temperature in weather. We typically have four seasons per year, with the temperature being highest in summer. So if you plotted weather over several years, you'd see peaks happening every four seasons, giving us the concept of seasonality. But this phenomenon isn't limited to weather—consider, for example, [Figure 9-3](#), which is a plot of traffic to a website.

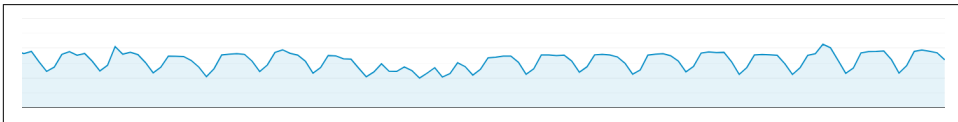


Figure 9-3. Website traffic

It's plotted week by week, and you can see regular dips. Can you guess what they are? The site in this case is one that provides information for software developers, and as you would expect, it gets less traffic on weekends! Thus, the time series has a seasonality of five high days and two low days. The data is plotted over several months, with the Christmas and New Year's holidays roughly in the middle, so you can see an additional seasonality there. If I had plotted it over some years, you'd clearly see the additional end-of-year dip.

There are many ways that seasonality can manifest in a time series. Traffic to a retail website, for instance, might peak on the weekends.

Autocorrelation

Another feature that you may see in time series is when there's predictable behavior after an event. You can see this in [Figure 9-4](#), where there are clear spikes, but after each spike, there's a deterministic decay. This is called *autocorrelation*.

In this case, we can see a particular set of behavior, which is repeated. Autocorrelations may be hidden in a time series pattern, but they have inherent predictability, so a time series containing many of them may be predictable.

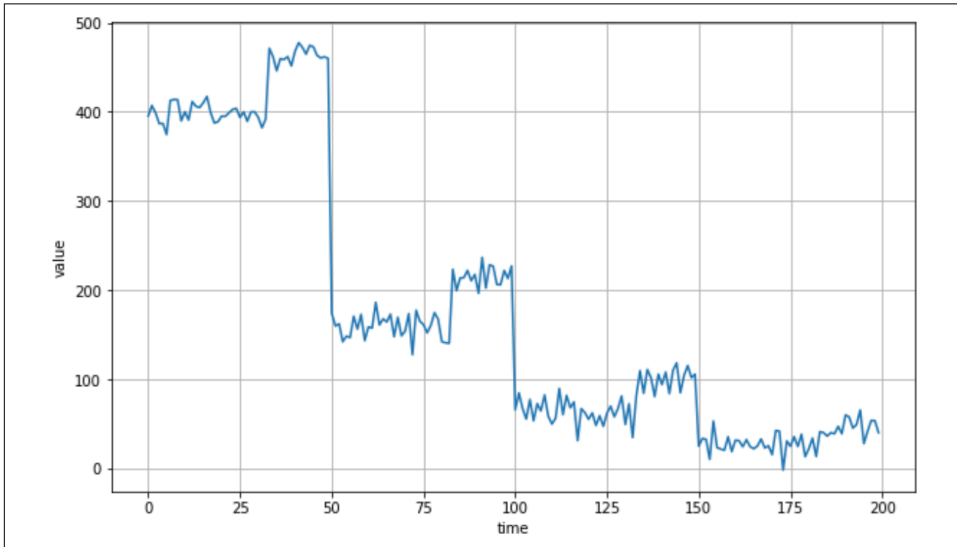


Figure 9-4. Autocorrelation

Noise

As its name suggests, noise is a set of seemingly random perturbations in a time series. These perturbations lead to a high level of unpredictability and can mask trends, seasonal behavior, and autocorrelation. For example, [Figure 9-5](#) shows the same autocorrelation from [Figure 9-4](#), but with a little noise added. Suddenly it's much harder to see the autocorrelation and predict values.

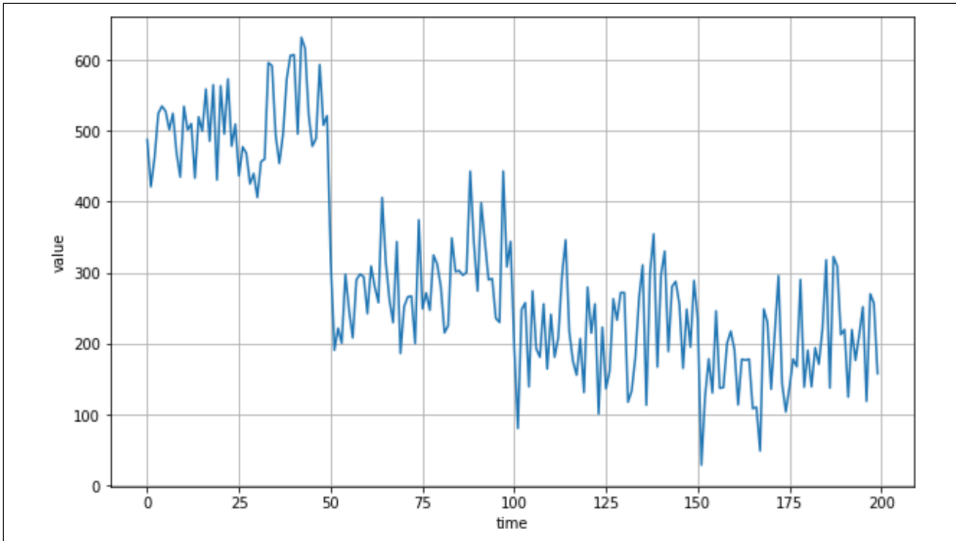


Figure 9-5. Autocorrelated series with added noise

Given all of these factors, let's explore how you can make predictions on time series that contain these attributes.

Techniques for Predicting Time Series

Before we get into ML-based prediction—the topic of the next few chapters—we'll explore some more naive prediction methods. These will enable you to establish a baseline that you can use to measure the accuracy of your ML predictions.

Naive Prediction to Create a Baseline

The most basic method to predict a time series is to say that the predicted value at time $t + 1$ is the same as the value from time t , effectively shifting the time series by a single period.

Let's begin by creating a time series that has trend, seasonality, and noise:

```
def plot_series(time, series, format="-", start=0, end=None):
    plt.plot(time[start:end], series[start:end], format)
    plt.xlabel("Time")
    plt.ylabel("Value")
    plt.grid(True)

def trend(time, slope=0):
    return slope * time

def seasonal_pattern(season_time):
```

```

"""Just an arbitrary pattern, you can change it if you wish"""
    return np.where(season_time < 0.4,
                    np.cos(season_time * 2 * np.pi),
                    1 / np.exp(3 * season_time))

def seasonality(time, period, amplitude=1, phase=0):
    """Repeats the same pattern at each period"""
    season_time = ((time + phase) % period) / period
    return amplitude * seasonal_pattern(season_time)

def noise(time, noise_level=1, seed=None):
    rnd = np.random.RandomState(seed)
    return rnd.randn(len(time)) * noise_level

time = np.arange(4 * 365 + 1, dtype="float32")
baseline = 10
series = trend(time, .05)
baseline = 10
amplitude = 15
slope = 0.09
noise_level = 6

# Create the series
series = baseline + trend(time, slope)
                + seasonality(time, period=365, amplitude=amplitude)
# Update with noise
series += noise(time, noise_level, seed=42)

```

After plotting this you'll see something like [Figure 9-6](#).

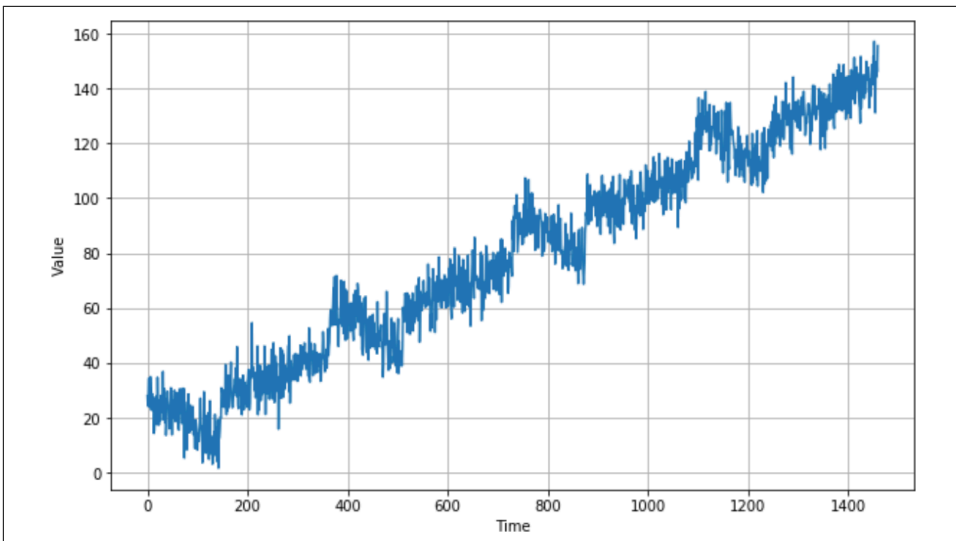


Figure 9-6. A time series showing trend, seasonality, and noise

Now that you have the data, you can split it like any data source into a training set, a validation set, and a test set. When there's some seasonality in the data, as you can see in this case, it's a good idea when splitting the series to ensure that there are whole seasons in each split. So, for example, if you wanted to split the data in [Figure 9-6](#) into training and validation sets, a good place to do this might be at time step 1,000, giving you training data up to step 1,000 and validation data after step 1,000.

You don't actually need to do the split here because you're just doing a naive forecast where each value t is simply the value at step $t - 1$, but for the purposes of illustration in the next few figures we'll zoom in on the data from time step 1,000 onwards.

To predict the series from a split time period onwards, where the period that you want to split from is in the variable `split_time`, you can use code like this:

```
naive_forecast = series[split_time - 1:-1]
```

[Figure 9-7](#) shows the validation set (from time step 1,000 onwards, which you get by setting `split_time` to 1000) with the naive prediction overlaid.

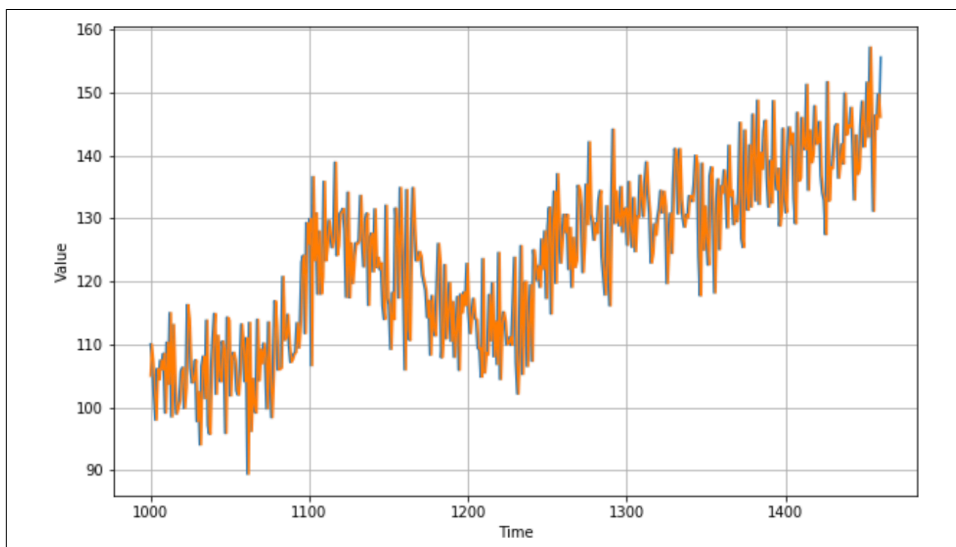


Figure 9-7. Naive forecast on time series

It looks pretty good—there is a relationship between the values—and, when charted over time, the predictions appear to closely match the original values. But how would you measure the accuracy?

Measuring Prediction Accuracy

There are a number of ways to measure prediction accuracy, but we'll concentrate on two of them: the *mean squared error* (MSE) and *mean absolute error* (MAE).

With MSE, you simply take the difference between the predicted value and the actual value at time t , square it (to remove negatives), and then find the average over all of them.

With MAE, you calculate the difference between the predicted value and the actual value at time t , take its absolute value to remove negatives (instead of squaring), and find the average over all of them.

For the naive forecast you just created based on our synthetic time series, you can get the MSE and MAE like this:

```
print(keras.metrics.mean_squared_error(x_valid, naive_forecast).numpy())
print(keras.metrics.mean_absolute_error(x_valid, naive_forecast).numpy())
```

I got an MSE of 76.47 and an MAE of 6.89. As with any prediction, if you can reduce the error, you can increase the accuracy of your predictions. We'll look at how to do that next.

Less Naive: Using Moving Average for Prediction

The previous naive prediction took the value at time $t - 1$ to be the forecasted value at time t . Using a moving average is similar, but instead of just taking the value from $t - 1$, it takes a group of values (say, 30), averages them out, and sets that to be the predicted value at time t . Here's the code:

```
def moving_average_forecast(series, window_size):
    """Forecasts the mean of the last few values.
    If window_size=1, then this is equivalent to naive forecast"""
    forecast = []
    for time in range(len(series) - window_size):
        forecast.append(series[time:time + window_size].mean())
    return np.array(forecast)

moving_avg = moving_average_forecast(series, 30)[split_time - 30:]

plt.figure(figsize=(10, 6))
plot_series(time_valid, x_valid)
plot_series(time_valid, moving_avg)
```

Figure 9-8 shows the plot of the moving average against the data.

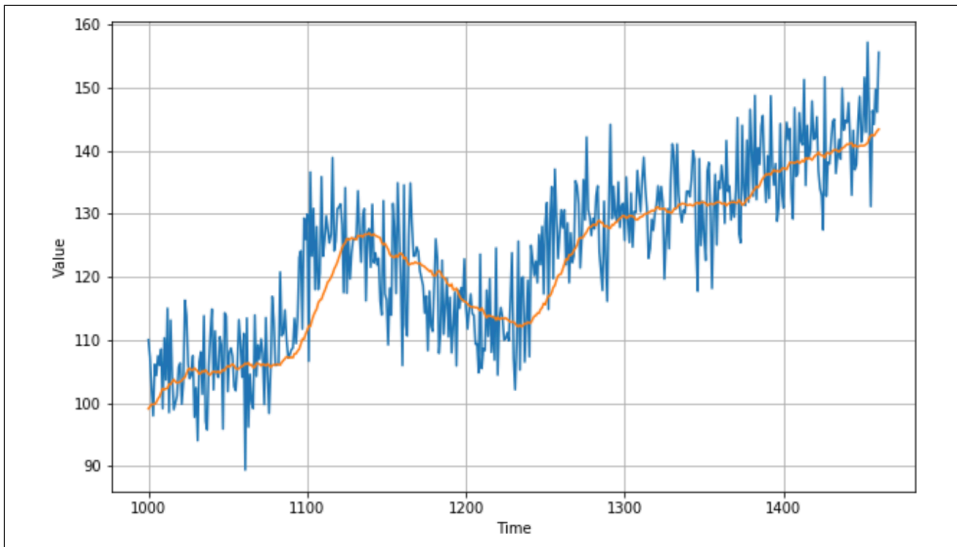


Figure 9-8. Plotting the moving average

When I plotted this time series I got an MSE and MAE of 49 and 5.5, respectively, so it's definitely improved the prediction a little. But this approach doesn't take into account the trend or the seasonality, so we may be able to improve it further with a little analysis.

Improving the Moving Average Analysis

Given that the seasonality in this time series is 365 days, you can smooth out the trend and seasonality using a technique called *differencing*, which just subtracts the value at $t - 365$ from the value at t . This will flatten out the diagram. Here's the code:

```
diff_series = (series[365:] - series[:-365])
diff_time = time[365:]
```

You can now calculate a moving average of *these* values and add back in the past values:

```
diff_moving_avg =
    moving_average_forecast(diff_series, 50)[split_time - 365 - 50:]

diff_moving_avg_plus_smooth_past =
    moving_average_forecast(series[split_time - 370:-360], 10) +
    diff_moving_avg
```

When you plot this (see [Figure 9-9](#)), you can already see an improvement in the predicted values: the trend line is very close to the actual values, albeit with the noise smoothed out. Seasonality seems to be working, as does the trend.

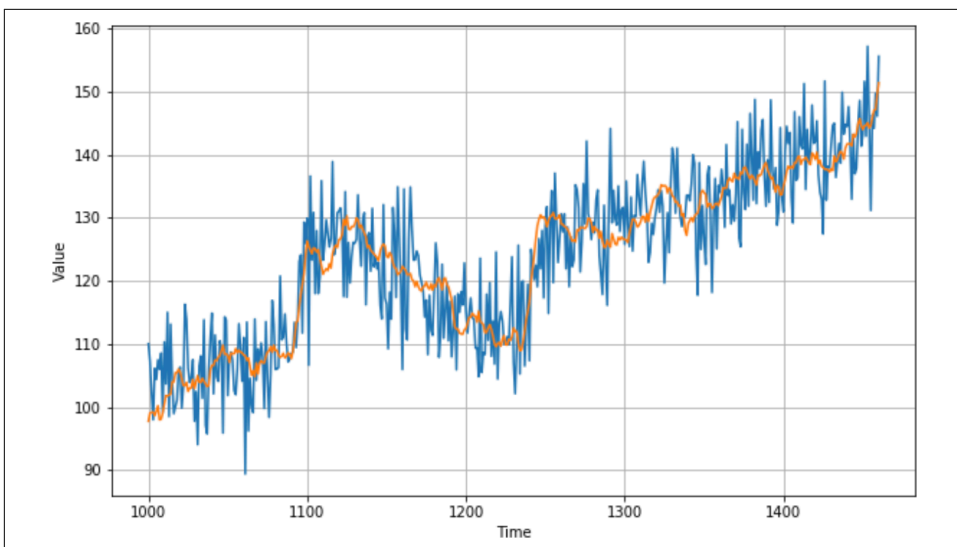


Figure 9-9. Improved moving average

This impression is confirmed by calculating the MSE and MAE—in this case I got 40.9 and 5.13, respectively, showing a clear improvement in the predictions.

Summary

This chapter introduced time series data and some of the common attributes of time series. You created a synthetic time series and saw how you can start making naive predictions on it. From these predictions, you established baseline measurements using mean squared error and mean average error. It was a nice break from TensorFlow, but in the next chapter you'll go back to using TensorFlow and ML to see if you can improve on your predictions!

Creating ML Models to Predict Sequences

Chapter 9 introduced sequence data and the attributes of a time series, including seasonality, trend, autocorrelation, and noise. You created a synthetic series to use for predictions and explored how to do basic statistical forecasting. Over the next couple of chapters, you'll learn how to use ML for forecasting. But before you start creating models, you need to understand how to structure the time series data for training predictive models, by creating what we'll call a *windowed dataset*.

To understand why you need to do this, consider the time series you created in Chapter 9. You can see a plot of it in Figure 10-1.

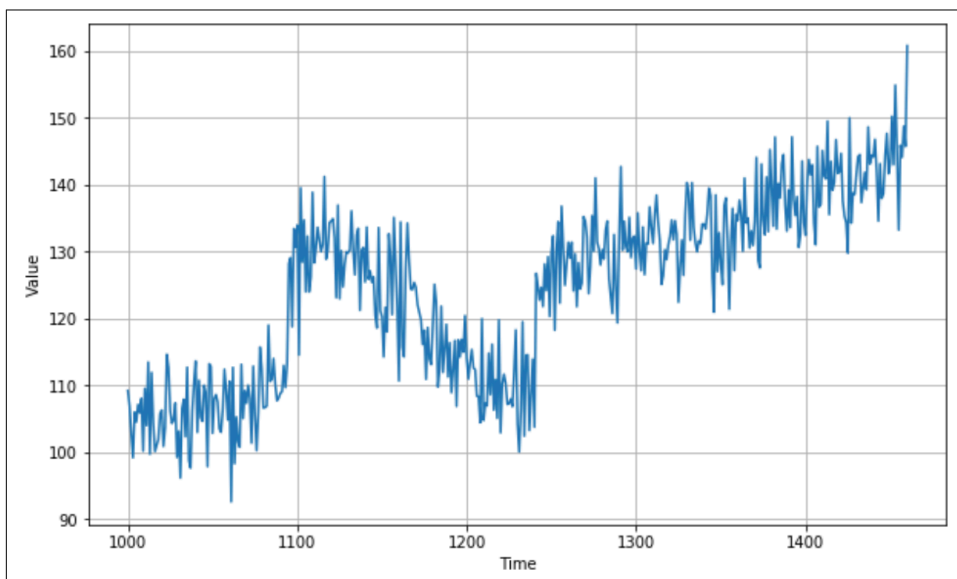


Figure 10-1. Synthetic time series

If at any point you want to predict a value at time t , you'll want to predict it as a function of the values preceding time t . For example, say you want to predict the value of the time series at time step 1,200 as a function of the 30 values preceding it. In this case, the values from time steps 1,170 to 1,199 would determine the value at time step 1,200, as shown in [Figure 10-2](#).

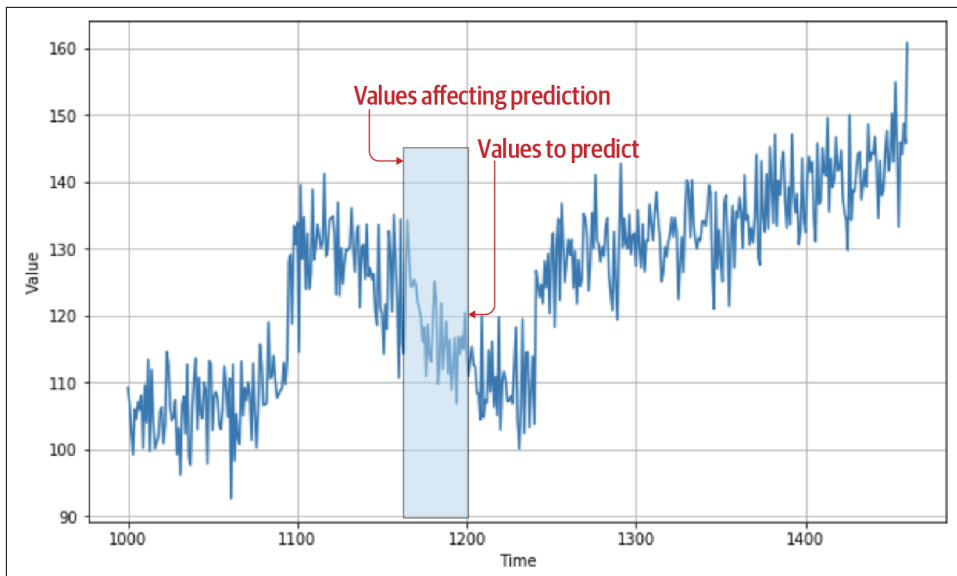


Figure 10-2. Previous values impacting prediction

Now this begins to look familiar: you can consider the values from 1,170–1,199 to be your *features* and the value at 1,200 to be your *label*. If you can get your dataset into a condition where you have a certain number of values as features and the following one as the label, and you do this for every known value in the dataset, you'll end up with a pretty decent set of features and labels that can be used to train a model.

Before doing this for the time series dataset from [Chapter 9](#), let's create a very simple dataset that has all the same attributes, but with a much smaller amount of data.

Creating a Windowed Dataset

The `tf.data` libraries contain a lot of APIs that are useful for manipulating data. You can use these to create a basic dataset containing the numbers 0–9, emulating a time series. You'll then turn that into the beginnings of a windowed dataset. Here's the code:

```
dataset = tf.data.Dataset.range(10)
dataset = dataset.window(5, shift=1, drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(5))
```

```
for window in dataset:
    print(window.numpy())
```

First it creates the dataset using a range, which simply makes the dataset contain the values 0 to $n - 1$, where n is, in this case, 10.

Next, calling `dataset.window` and passing a parameter of 5 specifies to split the dataset into windows of five items. Specifying `shift=1` causes each window to then be shifted one spot from the previous one: the first window will contain the five items beginning at 0, the next window the five items beginning at 1, etc. Setting `drop_remainder` to `True` specifies that once it reaches the point close to the end of the dataset where the windows would be smaller than the desired size of five, they should be dropped.

Given the window definition, the process of splitting the dataset can take place. You do this with the `flat_map` function, in this case requesting a batch of five windows.

Running this code will give the following result:

```
[0 1 2 3 4]
[1 2 3 4 5]
[2 3 4 5 6]
[3 4 5 6 7]
[4 5 6 7 8]
[5 6 7 8 9]
```

But earlier you saw that we want to make training data out of this, where there are n values defining a feature and a subsequent value giving a label. You can do this by adding another lambda function that splits each window into everything before the last value, and then the last value. This gives an `x` and a `y` dataset, as shown here:

```
dataset = tf.data.Dataset.range(10)
dataset = dataset.window(5, shift=1, drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(5))
dataset = dataset.map(lambda window: (window[:-1], window[-1:]))
for x,y in dataset:
    print(x.numpy(), y.numpy())
```

The results are now in line with what you'd expect. The first four values in the window can be thought of as the features, with the subsequent value being the label:

```
[0 1 2 3] [4]
[1 2 3 4] [5]
[2 3 4 5] [6]
[3 4 5 6] [7]
[4 5 6 7] [8]
[5 6 7 8] [9]
```

And because this is a dataset, it can also support shuffling and batching via lambda functions. Here, it's been shuffled and batched with a batch size of 2:

```

dataset = tf.data.Dataset.range(10)
dataset = dataset.window(5, shift=1, drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(5))
dataset = dataset.map(lambda window: (window[:-1], window[-1:]))
dataset = dataset.shuffle(buffer_size=10)
dataset = dataset.batch(2).prefetch(1)
for x,y in dataset:
    print("x = ", x.numpy())
    print("y = ", y.numpy())

```

The results show that the first batch has two sets of x (starting at 2 and 3, respectively) with their labels, the second batch has two sets of x (starting at 1 and 5, respectively) with their labels, and so on:

```

x = [[2 3 4 5]
     [3 4 5 6]]
y = [[6]
     [7]]

x = [[1 2 3 4]
     [5 6 7 8]]
y = [[5]
     [9]]

x = [[0 1 2 3]
     [4 5 6 7]]
y = [[4]
     [8]]

```

With this technique, you can now turn any time series dataset into a set of training data for a neural network. In the next section, you'll explore how to take the synthetic data from [Chapter 9](#) and create a training set from it. From there you'll move on to creating a simple DNN that is trained on this data and can be used to predict future values.

Creating a Windowed Version of the Time Series Dataset

As a recap, here's the code used in the previous chapter to create a synthetic time series dataset:

```

def trend(time, slope=0):
    return slope * time

def seasonal_pattern(season_time):
    return np.where(season_time < 0.4,
                    np.cos(season_time * 2 * np.pi),
                    1 / np.exp(3 * season_time))

def seasonality(time, period, amplitude=1, phase=0):
    season_time = ((time + phase) % period) / period
    return amplitude * seasonal_pattern(season_time)

```

```
def noise(time, noise_level=1, seed=None):
    rnd = np.random.RandomState(seed)
    return rnd.randn(len(time)) * noise_level

time = np.arange(4 * 365 + 1, dtype="float32")
series = trend(time, 0.1)
baseline = 10
amplitude = 20
slope = 0.09
noise_level = 5

series = baseline + trend(time, slope)
series += seasonality(time, period=365, amplitude=amplitude)
series += noise(time, noise_level, seed=42)
```

This will create a time series that looks like [Figure 10-1](#). If you want to change it, feel free to tweak the values of the various constants.

Once you have the series, you can turn it into a windowed dataset with code similar to that in the previous section. Here it is, defined as a standalone function:

```
def windowed_dataset(series, window_size,
                    batch_size, shuffle_buffer):
    dataset = tf.data.Dataset.from_tensor_slices(series)
    dataset = dataset.window(window_size + 1, shift=1,
                           drop_remainder=True)
    dataset = dataset.flat_map(lambda window:
                              window.batch(window_size + 1))
    dataset = dataset.shuffle(shuffle_buffer).map(
        lambda window:
            (window[:-1], window[-1]))
    dataset = dataset.batch(batch_size).prefetch(1)
    return dataset
```

Note that it uses the `from_tensor_slices` method of `tf.data.Dataset`, which allows you to turn a series into a `Dataset`. You can learn more about this method in the [TensorFlow documentation](#).

Now, to get a training-ready dataset you can simply use the following code. First you split the series into training and validation datasets, then you specify details like the size of the window, the batch size, and the shuffle buffer size:

```
split_time = 1000
time_train = time[:split_time]
x_train = series[:split_time]
time_valid = time[split_time:]
x_valid = series[split_time:]
window_size = 20
batch_size = 32
shuffle_buffer_size = 1000
```



```
dataset = windowed_dataset(x_train, window_size, batch_size,
                           shuffle_buffer_size)
```

The important thing to remember now is that your data is a `tf.data.Dataset`, so it can easily be passed to `model.fit` as a single parameter and `tf.keras` will take care of the rest.

If you want to inspect what the data looks like, you can do so with code like this:

```
dataset = windowed_dataset(series, window_size, 1, shuffle_buffer_size)
for feature, label in dataset.take(1):
    print(feature)
    print(label)
```

Here the `batch_size` is set to 1, just to make the results more readable. You'll end up with output like this, where a single set of data is in the batch:

```
tf.Tensor(
[[75.38214 66.902626 76.656364 71.96795 71.373764 76.881065 75.62607
 71.67851 79.358665 68.235466 76.79933 76.764114 72.32991 75.58744
 67.780426 78.73544 73.270195 71.66057 79.59881 70.9117 ]],
 shape=(1, 20), dtype=float32)
tf.Tensor([67.47085], shape=(1,), dtype=float32)
```

The first batch of numbers are the features. We set the window size to 20, so it's a 1×20 tensor. The second number is the label (67.47085 in this case), which the model will try to fit the features to. You'll see how that works in the next section.

Creating and Training a DNN to Fit the Sequence Data

Now that you have the data in a `tf.data.Dataset`, creating a neural network model in `tf.keras` becomes very straightforward. Let's first explore a simple DNN that looks like this:

```
dataset = windowed_dataset(series, window_size,
                           batch_size, shuffle_buffer_size)

model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(10, input_shape=[window_size],
                           activation="relu"),
    tf.keras.layers.Dense(10, activation="relu"),
    tf.keras.layers.Dense(1)
])
```

It's a super simple model with two dense layers, the first of which accepts the input shape of `window_size` before an output layer that will contain the predicted value.

The model is compiled with a loss function and optimizer, as before. In this case the loss function is specified as `mse`, which stands for mean squared error and is commonly used in regression problems (which is what this ultimately boils down to!). For the optimizer, `sgd` (stochastic gradient descent) is a good fit. I won't go into detail on

these types of functions in this book, but any good resource on machine learning will teach you about them—Andrew Ng’s seminal [Deep Learning Specialization](#) on Coursera is a great place to start. SGD takes parameters for learning rate (`lr`) and momentum, and these tweak how the optimizer learns. Every dataset is different, so it’s good to have control. In the next section, you’ll see how you can figure out the optimal values, but, for now, just set them like this:

```
model.compile(loss="mse",optimizer=tf.keras.optimizers.SGD(
                                                    lr=1e-6,
                                                    momentum=0.9))
```

Training then becomes as simple as calling `model.fit`, passing it your dataset, and specifying the number of epochs to train for:

```
model.fit(dataset,epochs=100,verbose=1)
```

As you train, you’ll see the loss function report a number that starts high but will decline steadily. Here’s the result of the first 10 epochs:

```
Epoch 1/100
45/45 [=====] - 1s 15ms/step - loss: 898.6162
Epoch 2/100
45/45 [=====] - 0s 8ms/step - loss: 52.9352
Epoch 3/100
45/45 [=====] - 0s 8ms/step - loss: 49.9154
Epoch 4/100
45/45 [=====] - 0s 7ms/step - loss: 49.8471
Epoch 5/100
45/45 [=====] - 0s 7ms/step - loss: 48.9934
Epoch 6/100
45/45 [=====] - 0s 7ms/step - loss: 49.7624
Epoch 7/100
45/45 [=====] - 0s 8ms/step - loss: 48.3613
Epoch 8/100
45/45 [=====] - 0s 9ms/step - loss: 49.8874
Epoch 9/100
45/45 [=====] - 0s 8ms/step - loss: 47.1426
Epoch 10/100
45/45 [=====] - 0s 8ms/step - loss: 47.5133
```

Evaluating the Results of the DNN

Once you have a trained DNN, you can start predicting with it. But remember, you have a windowed dataset, so, the prediction for a given point is based on the values of a certain number of time steps before it.

In other words, as your data is in a list called `series`, to predict a value you have to pass the model values from time t to time $t+\text{window_size}$. It will then give you the predicted value for the next time step.

For example, if you wanted to predict the value at time step 1,020, you would take the values from time steps 1,000 through 1,019 and use them to predict the next value in the sequence. To get those values, you use the following code (note that you specify this as `series[1000:1020]`, not `series[1000:1019]!`):

```
print(series[1000:1020])
```

Then, to get the value at step 1,020, you simply use `series[1020]` like this:

```
print(series[1020])
```

To get the prediction for that data point, you then pass the series into `model.predict`. Note, however, that in order to keep the input shape consistent you'll need an `[np.newaxis]`, like this:

```
print(model.predict(series[1000:1020][np.newaxis]))
```

Or, if you want code that's a bit more generic, you can use this:

```
print(series[start_point:start_point+window_size])
print(series[start_point+window_size])
print(model.predict(
    series[start_point:start_point+window_size][np.newaxis]))
```

Do note that all of this is assuming a window size of 20 data points, which is quite small. As a result, your model may lack some accuracy. If you want to try a different window size, you'll need to reformat the dataset by calling the `windowed_dataset` function again and then retraining the model.

Here is the output for this dataset when taking a start point of 1,000 and predicting the next value:

```
[109.170746 106.86935 102.61668 99.15634 105.95478 104.503876
 107.08533 105.858284 108.00339 100.15279 109.4894 103.96404
 113.426094 99.67773 111.87749 104.26137 100.08899 101.00105
 101.893265 105.69048 ]

106.258606

[[105.36248]]
```

The first tensor contains the list of values. Next, we see the *actual* next value, which is 106.258606. Finally, we see the *predicted* next value, 105.36248. We're getting a reasonable prediction, but how do we measure the accuracy over time? We'll explore that in the next section.

Exploring the Overall Prediction

In the previous section, you saw how to get a prediction for a given point in time by taking the previous set of values based on the window size (in this case 20) and passing them to the model. To see the overall results of the model you'll have to do the same for every time step.

You can do this with a simple loop like this:

```
forecast = []
for time in range(len(series) - window_size):
    forecast.append(
        model.predict(series[time:time + window_size][np.newaxis]))
```

First, you create a new array called `forecast` that will store the predicted values. Then, for every time step in the original series, you call the `predict` method and store the results in the `forecast` array. You can't do this for the first n elements in the data, where n is the `window_size`, because at that point you won't have enough data to make a prediction, as every prediction requires n previous values.

When this loop is finished, the `forecast` array will have the values of the predictions for time step 21 onwards.

If you recall, you also split the dataset into training and validation sets at time step 1,000. So, for the next step you should also only take the forecasts from this time onwards. As your forecast data is already off by 20 (or whatever your window size is), you can split it and turn it into a Numpy array like this:

```
forecast = forecast[split_time-window_size:]
results = np.array(forecast)[:, 0, 0]
```

It's now in the same shape as the prediction data, so you can plot them against each other like this:

```
plt.figure(figsize=(10, 6))

plot_series(time_valid, x_valid)
plot_series(time_valid, results)
```

The plot will look something like [Figure 10-3](#).

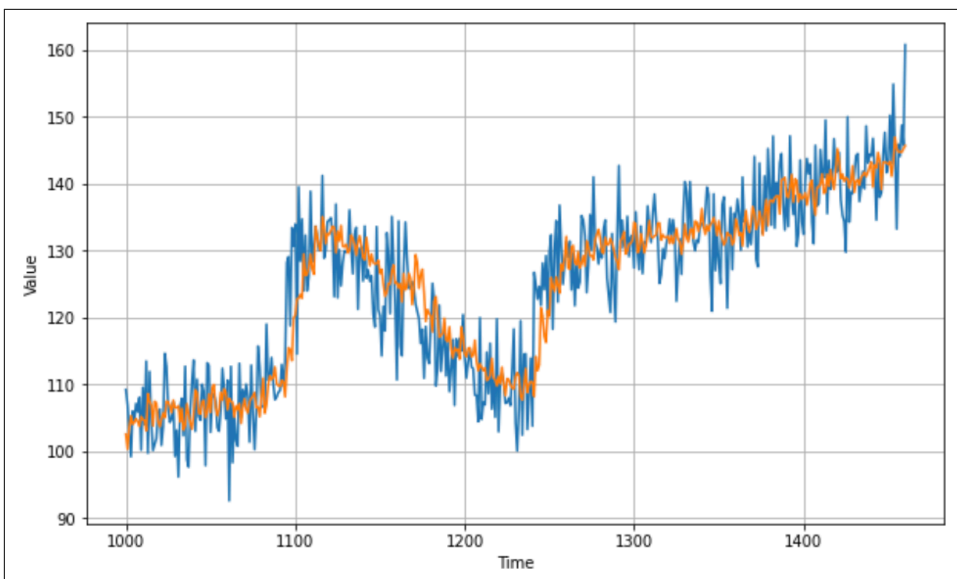


Figure 10-3. Plotting predictions against values

From a quick visual inspection, you can see that the prediction isn't bad. It's generally following the curve of the original data. When there are rapid changes in the data the prediction takes a little time to catch up, but on the whole it isn't bad.

However, it's hard to be precise when eyeballing the curve. It's best to have a good metric, and in [Chapter 9](#) you learned about one—the MAE. Now that you have the valid data and the results, you can measure the MAE with this code:

```
tf.keras.metrics.mean_absolute_error(x_valid, results).numpy()
```

There was randomness introduced to the data, so your results may vary, but when I tried it I got a value of 4.51 as the MAE.

You could argue that the process of getting the predictions as accurate as possible then becomes the process of minimizing that MAE. There are some techniques that you can use to do this, including the obvious changing of the window size. I'll leave you to experiment with that, but in the next section you'll do some basic hyperparameter tuning on the optimizer to improve how your neural network learns, and see what impact that will have on the MAE.

Tuning the Learning Rate

In the previous example, you might recall that you compiled the model with an optimizer that looked like this:

```
model.compile(loss="mse",
              optimizer=tf.keras.optimizers.SGD(lr=1e-6, momentum=0.9))
```

In this case, you used a learning rate of 1×10^{-6} . But that seemed to be a really arbitrary number. What if you changed it? And how should you go about changing it? It would take a lot of experimentation to find the best rate.

One thing that `tf.keras` offers you is a callback that helps you adjust the learning rate over time. You learned about callbacks—functions that are called at the end of every epoch—way back in [Chapter 2](#), where you used one to cancel training when the accuracy reached a desired value.

You can also use a callback to adjust the learning rate parameter, plot the value of that parameter against the loss for the appropriate epoch, and from there determine the best learning rate to use.

To do this, simply create a `tf.keras.callbacks.LearningRateScheduler` and have it populate the `lr` parameter with the desired starting value. Here's an example:

```
lr_schedule = tf.keras.callbacks.LearningRateScheduler(
    lambda epoch: 1e-8 * 10**(epoch / 20))
```

In this case you're going to start the learning rate at $1e-8$, and then every epoch increase it by a small amount. By the time it's completed one hundred epochs, the learning rate will be up to about $1e-3$.

Now you can initialize the optimizer with the learning rate of $1e-8$, and specify that you want to use this callback within the `model.fit` call:

```
optimizer = tf.keras.optimizers.SGD(lr=1e-8, momentum=0.9)
model.compile(loss="mse", optimizer=optimizer)
history = model.fit(dataset, epochs=100,
                   callbacks=[lr_schedule], verbose=0)
```

As you used `history=model.fit`, the training history is stored for you, including the loss. You can then plot this against the learning rate per epoch like this:

```
lrs = 1e-8 * (10 ** (np.arange(100) / 20))
plt.semilogx(lrs, history.history["loss"])
plt.axis([1e-8, 1e-3, 0, 300])
```

This just sets the `lrs` value using the same formula as the `lambda` function, and plots this against loss between $1e-8$ and $1e-3$. [Figure 10-4](#) shows the result.

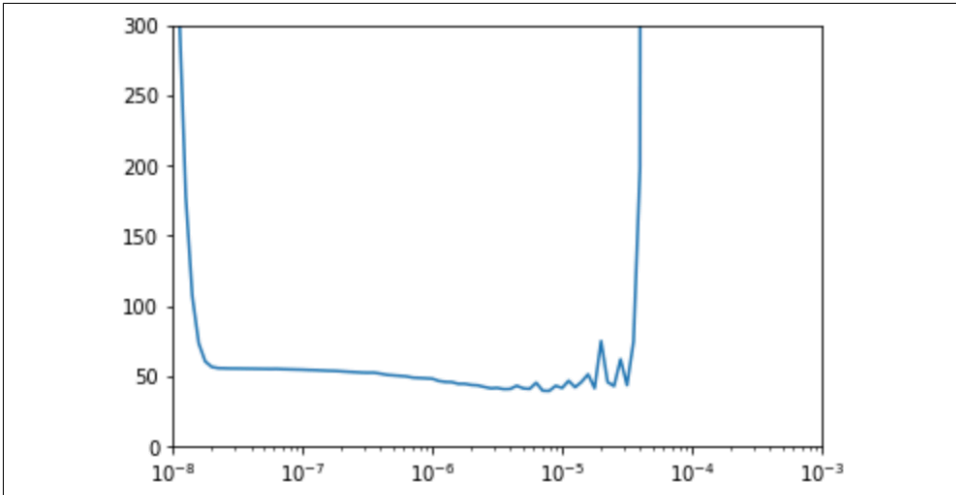


Figure 10-4. Plotting loss versus learning rate

So, while earlier you had set the learning rate to $1e-6$, it looks like $1e-5$ has a smaller loss, so now you can go back to the model and redefine it with $1e-5$ as the new learning rate.

After training the model, you'll likely notice that the loss has reduced a bit. In my case, with the learning rate at $1e-6$ my final loss was 36.5, but with the learning rate at $1e-5$ it was reduced to 32.9. However, when I ran predictions for all the data, the result was the chart in [Figure 10-5](#), which as you can see looks a little off.

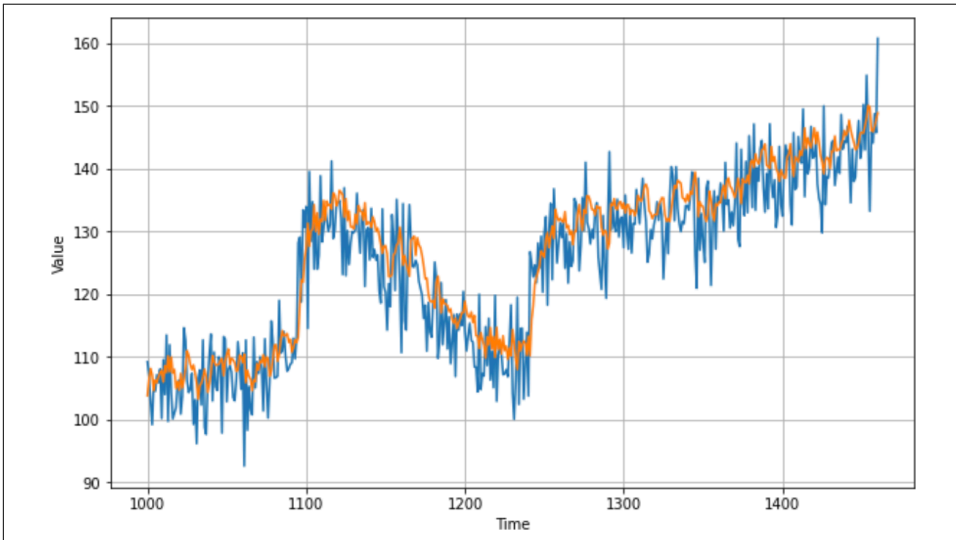


Figure 10-5. Chart with the adjusted learning rate

And when I measured the MAE it ended up as 4.96, so it has taken a small step backward! That being said, once you know you have the best learning rate, you can start exploring other methodologies to optimize the network's performance. An easy place to begin is with the size of the window—20 days data to predict 1 day may not be enough, so you might want to try a window of 40 days. Also, try training for more epochs. With a bit of experimentation you could get an MAE of close to 4, which isn't bad.

Exploring Hyperparameter Tuning with Keras Tuner

In the previous section, you saw how to do a rough optimization of the learning rate for the stochastic gradient descent loss function. It was certainly a very rough effort, changing the learning rate every few epochs and measuring the loss. It also was somewhat tainted by the fact that the loss function was *already* changing epoch to epoch, so you may not have actually been finding the best value, but an approximation. To really find the best value, you would have to train for the full set of epochs with each potential value, and then compare the results. And that's just for one hyperparameter, the learning rate. If you want to find the best momentum, or tune other things like the model architecture—how many neurons per layer, how many layers, etc.—you can end up with many thousands of options to test, and training across all of these would be hard to code.

Fortunately, the **Keras Tuner tool** makes this relatively easy. You can install Keras Tuner using a simple `pip` command:

```
!pip install keras-tuner
```

You can then use it to parameterize your hyperparameters, specifying ranges of values to test. Keras Tuner will train multiple models, one with each possible set of parameters, evaluate the model to a metric you want, and then report on the top models. I won't go into all of the options the tool offers here, but I will show how you can use it for this specific model.

Say we want to experiment with just two things, the first being the number of input neurons in the model architecture. All along you've had a model architecture of 10 input neurons, followed by a hidden layer of 10, followed by the output layer. But could the network do better with more? What if you could experiment with up to 30 neurons in the input layer, for example?

Recall that the input layer was defined like this:

```
tf.keras.layers.Dense(10, input_shape=[window_size], activation="relu"),
```

If you want to test different values than the hardcoded 10, you can set it to cycle through a number of integers like this:


```
tf.keras.layers.Dense(units=hp.Int('units', min_value=10, max_value=30, step=2),
                       activation='relu', input_shape=[window_size])
```

Here you define that the layer will be tested with several input values, starting with 10 and increasing to 30 in steps of 2. Now, instead of training the model just once and seeing the loss, Keras Tuner will train the model 11 times!

Also, when you compiled the model, you hardcoded the value of the momentum parameter to be 0.9. Recall this code from the model definition:

```
optimizer = tf.keras.optimizers.SGD(lr=1e-5, momentum=0.9)
```

You can change this to cycle through a few options instead by using the `hp.Choice` construct. Here's an example:

```
optimizer=tf.keras.optimizers.SGD(hp.Choice('momentum',
                                             values=[.9, .7, .5, .3]),
                                  lr=1e-5)
```

This provides four possible choices, so, when combined with the model architecture defined previously you'll end up cycling through 44 possible combinations. Keras Tuner can do this for you, and report back on the model that performed the best.

To finish setting this up, first create a function that builds the model for you. Here's an updated model definition:

```
def build_model(hp):
    model = tf.keras.models.Sequential()
    model.add(tf.keras.layers.Dense(
        units=hp.Int('units', min_value=10, max_value=30, step=2),
        activation='relu', input_shape=[window_size]))
    model.add(tf.keras.layers.Dense(10, activation='relu'))
    model.add(tf.keras.layers.Dense(1))

    model.compile(loss="mse",
                  optimizer=tf.keras.optimizers.SGD(hp.Choice('momentum',
                                                                values=[.9, .7, .5, .3]),
                                                                lr=1e-5))

    return model
```

Now, with Keras Tuner installed, you can create a `RandomSearch` object that manages all the iterations for this model:

```
tuner = RandomSearch(build_model,
                     objective='loss', max_trials=150,
                     executions_per_trial=3, directory='my_dir',
                     project_name='hello')
```

Note that you define the model by passing it the function that you described earlier. The hyperparameter parameter (`hp`) is used to control which values get changed. You specify the objective to be `loss`, indicating that you want to minimize the loss. You

can cap the overall number of trials to run with the `max_trials` parameter, and specify how many times to train and evaluate the model (eliminating random fluctuations somewhat) with the `executions_per_trial` parameter.

To start the search, you simply call `tuner.search` like you would `model.fit`. Here's the code:

```
tuner.search(dataset, epochs=100, verbose=0)
```

Running this with the synthetic series you've been working on in this chapter will then train models with every possible hyperparameter according to your definition of the options you want to try.

When it's complete, you can call `tuner.results_summary` and it will give you your top 10 trials based on the objective:

```
tuner.results_summary()
```

You should see output like this:

```
Results summary
|-Results in my_dir/hello
|-Showing 10 best trials
|-Objective(name='loss', direction='min')
Trial summary
|-Trial ID: dcf832e62daf4d34b729c546120fb14
|-Score: 33.18723194615371
|-Best step: 0
Hyperparameters:
|-momentum: 0.5
|-units: 28
Trial summary
|-Trial ID: 02ca5958ac043f6be8b2e2b5479d1f09
|-Score: 33.83273440510237
|-Best step: 0
Hyperparameters:
|-momentum: 0.7
|-units: 28
```

From the results, you can see that the best loss score was achieved with a momentum of 0.5 and 28 input units. You can retrieve this model and other top models by calling `get_best_models` and specifying how many you want—for example, if you want the top four models you'd call it like this:

```
tuner.get_best_models(num_models=4)
```

You can then test those models.

Alternatively, you could create a new model from scratch using the learned hyperparameters, like this:

```
dataset = windowed_dataset(x_train, window_size, batch_size,
                           shuffle_buffer_size)

model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(28, input_shape=[window_size],
                          activation="relu"),
    tf.keras.layers.Dense(10, activation="relu"),
    tf.keras.layers.Dense(1)
])

optimizer = tf.keras.optimizers.SGD(lr=1e-5, momentum=0.5)
model.compile(loss="mse", optimizer=optimizer)
history = model.fit(dataset, epochs=100, verbose=1)
```

When I trained using these hyperparameters, and forecasting for the entire validation set as earlier, I got a chart that looked like [Figure 10-6](#).

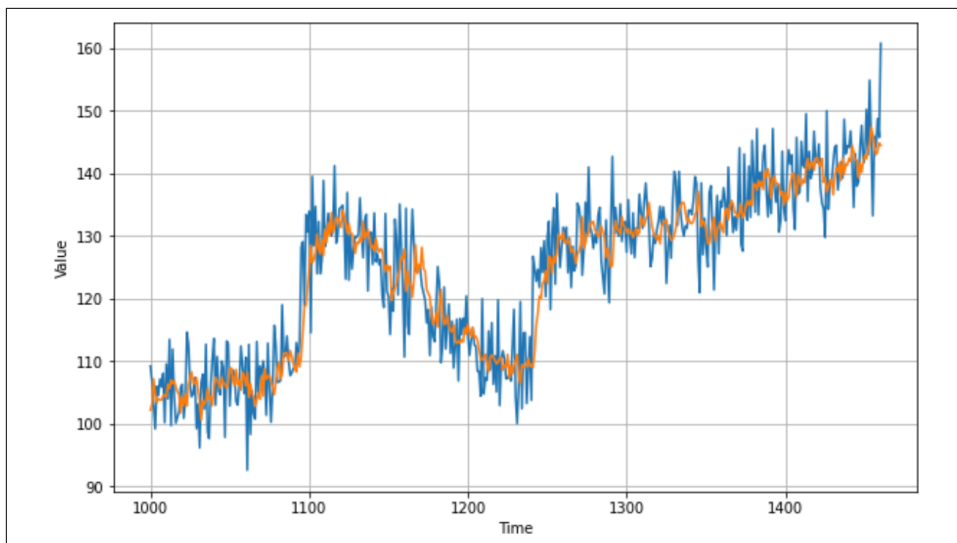


Figure 10-6. The prediction chart with optimized hyperparameters

A calculation of the MAE on this yielded 4.47, which is a slight improvement on the original of 4.51 and a big improvement over the last chapter's statistical approach that gave me a result of 5.13. This was done with the learning rate changed to 1e-5, which may not have been optimal. Using Keras Tuner you can tweak hyperparameters like that, adjust the number of neurons in the middle layer, or even experiment with different loss functions and optimizers. Give it a try and see if you can improve this model!

Summary

In this chapter, you took the statistical analysis of the time series from [Chapter 9](#) and applied machine learning to try to do a better job of prediction. Machine learning really is all about pattern matching, and, as expected, you were able to take the mean average error down almost 10% by first using a deep neural network to spot the patterns, and then using hyperparameter tuning with Keras Tuner to improve the loss and increase the accuracy. In [Chapter 11](#), you'll go beyond a simple DNN and examine the implications of using a recurrent neural network to predict sequential values.

Using Convolutional and Recurrent Methods for Sequence Models

The last few chapters introduced you to sequence data. You saw how to predict it first using statistical methods, then basic machine learning methods with a deep neural network. You also explored how to tune the model's hyperparameters using Keras Tuner. In this chapter, you'll look at additional techniques that may further enhance your ability to predict sequence data using convolutional neural networks as well as recurrent neural networks.

Convolutions for Sequence Data

In [Chapter 3](#) you were introduced to convolutions where a 2D filter was passed over an image to modify it and potentially extract features. Over time, the neural network learned which filter values were effective at matching the modifications made to the pixels to their labels, effectively extracting features from the image. The same technique can be applied to numeric time series data, but with one modification: the convolution will be one-dimensional instead of two-dimensional.

Consider, for example, the series of numbers in [Figure 11-1](#).

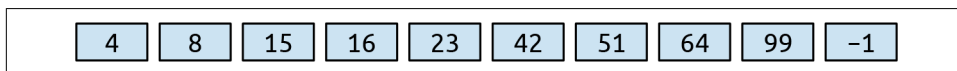


Figure 11-1. A sequence of numbers

A 1D convolution could operate on these as follows. Consider the convolution to be a 1×3 filter with filter values of -0.5 , 1 , and -0.5 , respectively. In this case, the first value in the sequence will be lost, and the second value will be transformed from 8 to -1.5 , as shown in [Figure 11-2](#).

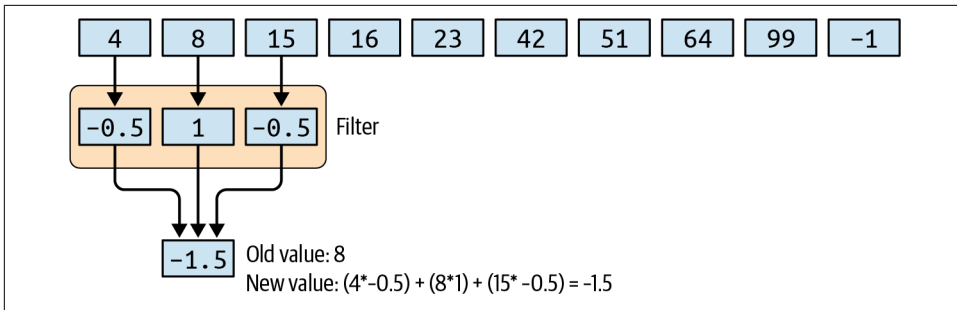


Figure 11-2. Using a convolution with the number sequence

The filter will then stride across the values, calculating new ones as it goes. So, for example, in the next stride 15 will be transformed to 3, as shown in Figure 11-3.

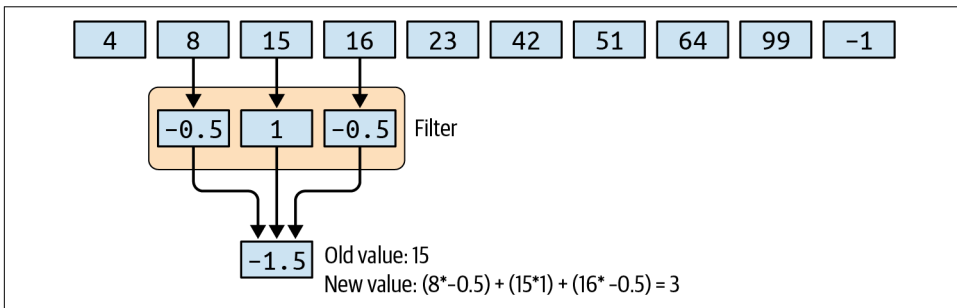


Figure 11-3. An additional stride in the 1D convolution

Using this method, it's possible to extract the patterns between values and learn the filters that extract them successfully, in much the same way as convolutions on the pixels in images are able to extract features. In this instance there are no labels, but the convolutions that minimize overall loss could be learned.

Coding Convolutions

Before coding convolutions, you'll have to adjust the windowed dataset generator that you used in the previous chapter. This is because when coding the convolutional layers, you have to specify the dimensionality. The windowed dataset was a single dimension, but it wasn't defined as a 1D tensor. This simply requires adding a `tf.expand_dims` statement at the beginning of the `windowed_dataset` function like this:

```
def windowed_dataset(series, window_size, batch_size, shuffle_buffer):
    series = tf.expand_dims(series, axis=-1)
    dataset = tf.data.Dataset.from_tensor_slices(series)
    dataset = dataset.window(window_size + 1, shift=1, drop_remainder=True)
    dataset = dataset.flat_map(lambda window: window.batch(window_size + 1))
```

```
dataset = dataset.shuffle(shuffle_buffer).map(
    lambda window: (window[:-1], window[-1]))
dataset = dataset.batch(batch_size).prefetch(1)
return dataset
```

Now that you have an amended dataset, you can add a convolutional layer before the dense layers that you had previously:

```
dataset = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)

model = tf.keras.models.Sequential([
    tf.keras.layers.Conv1D(filters=128, kernel_size=3,
                           strides=1, padding="causal",
                           activation="relu",
                           input_shape=[None, 1]),
    tf.keras.layers.Dense(28, activation="relu"),
    tf.keras.layers.Dense(10, activation="relu"),
    tf.keras.layers.Dense(1),
])

optimizer = tf.keras.optimizers.SGD(lr=1e-5, momentum=0.5)
model.compile(loss="mse", optimizer=optimizer)
history = model.fit(dataset, epochs=100, verbose=1)
```

In the Conv1D layer, you have a number of parameters:

filters

Is the number of filters that you want the layer to learn. It will generate this number, and adjust them over time to fit your data as it learns.

kernel_size

Is the size of the filter—earlier we demonstrated a filter with the values -0.5 , 1 , -0.5 , which would be a kernel size of 3 .

strides

Is the size of the “step” that the filter will take as it scans across the list. This is typically 1 .

padding

Determines the behavior of the list with regard to which end data is dropped from. A 3×1 filter will “lose” the first and last value of the list because it cannot calculate the prior value for the first, or the subsequent value for the last. Typically with sequence data you’ll use `causal` here, which will only take data from the current and previous time steps, never future ones. So, for example, a 3×1 filter would take the current time step along with the previous two.

activation

Is the activation function. In this case, `relu` means to effectively reject negative values coming out of the layer.

`input_shape`

As always, is the input shape of the data being passed into the network. As this is the first layer, you have to specify it.

Training with this will give you a model as before, but to get predictions from the model, given that the input layer has changed shape, you'll need to modify your prediction code somewhat.

Also, instead of predicting each value, one by one, based on the previous window, you can actually get a single prediction for an entire series if you've correctly formatted the series as a dataset. To simplify things a bit, here's a helper function that can predict an entire series based on the model, with a specified window size:

```
def model_forecast(model, series, window_size):
    ds = tf.data.Dataset.from_tensor_slices(series)
    ds = ds.window(window_size, shift=1, drop_remainder=True)
    ds = ds.flat_map(lambda w: w.batch(window_size))
    ds = ds.batch(32).prefetch(1)
    forecast = model.predict(ds)
    return forecast
```

If you want to use the model to predict this series, you simply pass the series in with a new axis to handle the Conv1Ds needed for the layer with the extra axis. You can do it like this:

```
forecast = model_forecast(model, series[..., np.newaxis], window_size)
```

And you can split this forecast into just the predictions for the validation set using the predetermined split time:

```
results = forecast[split_time - window_size:-1, -1, 0]
```

A plot of the results against the series is in [Figure 11-4](#).

The MAE in this case is 4.89, which is slightly worse than for the previous prediction. This could be because we haven't tuned the convolutional layer appropriately, or it could be that convolutions simply don't help. This is the type of experimentation you'll need to do with your data.

Do note that this data has a random element in it, so values will change across sessions. If you're using code from [Chapter 10](#) and then running this code separately, you will, of course, have random fluctuations affecting your data, and thus your MAE.

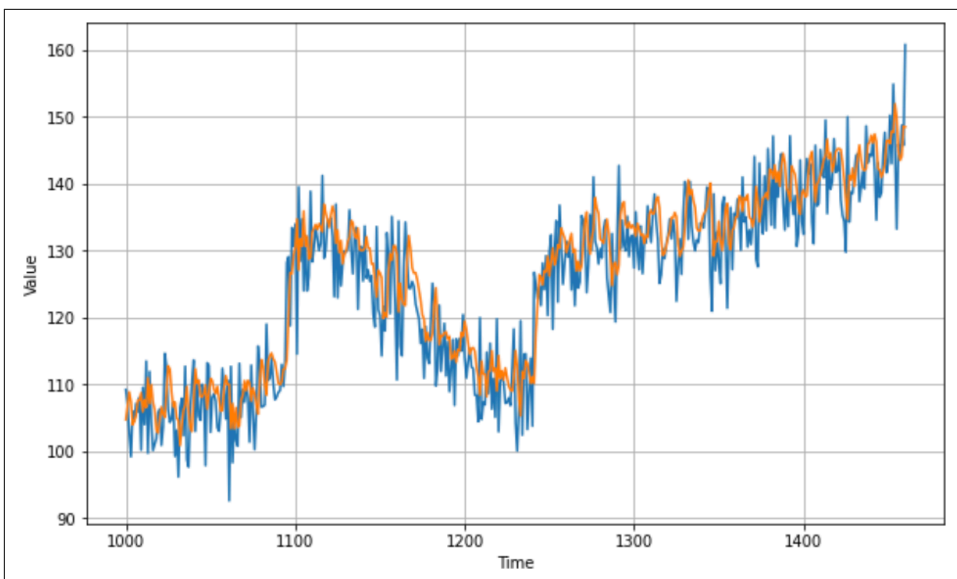


Figure 11-4. Convolutional neural network with time sequence data prediction

But when using convolutions, the question always comes up: Why choose the parameters that we chose? Why 128 filters? Why size 3×1 ? The good news is that you can experiment with them using **Keras Tuner**, as shown previously. We'll explore that next.

Experimenting with the Conv1D Hyperparameters

In the previous section, you saw a 1D convolution that was hardcoded with parameters for things like filter number, kernel size, number of strides, etc. When training the neural network with it, it appeared that the MAE went up slightly, so we got no benefit from using the Conv1D. This may not always be the case, depending on your data, but it could be because of suboptimal hyperparameters. So, in this section, you'll see how Keras Tuner can optimize them for you.

In this example, you'll experiment with the hyperparameters for the number of filters, the size of the kernel, and the size of the stride, keeping the other parameters static:

```
def build_model(hp):
    model = tf.keras.models.Sequential()
    model.add(tf.keras.layers.Conv1D(
        filters=hp.Int('units', min_value=128, max_value=256, step=64),
        kernel_size=hp.Int('kernels', min_value=3, max_value=9, step=3),
        strides=hp.Int('strides', min_value=1, max_value=3, step=1),
        padding='causal', activation='relu', input_shape=[None, 1]
    ))
```

```

model.add(tf.keras.layers.Dense(28, input_shape=[window_size],
                                activation='relu'))

model.add(tf.keras.layers.Dense(10, activation='relu'))

model.add(tf.keras.layers.Dense(1))

model.compile(loss="mse",
              optimizer=tf.keras.optimizers.SGD(momentum=0.5, lr=1e-5))
return model

```

The filter values will start at 128 and then step upwards toward 256 in increments of 64. The kernel size will start at 3 and increase to 9 in steps of 3, and the strides will start at 1 and be stepped up to 3.

There are a lot of combinations of values here, so the experiment will take some time to run. You could also try other changes, like using a much smaller starting value for filters to see the impact they have.

Here's the code to do the search:

```

tuner = RandomSearch(build_model, objective='loss',
                    max_trials=500, executions_per_trial=3,
                    directory='my_dir', project_name='cnn-tune')

tuner.search_space_summary()

tuner.search(dataset, epochs=100, verbose=2)

```

When I ran the experiment, I discovered that 128 filters, with size 9 and stride 1, gave the best results. So, compared to the initial model, the big difference was to change the filter size—which makes sense with such a large body of data. With a filter size of 3, only the immediate neighbors have an impact, whereas with 9, neighbors further afield also have an impact on the result of applying the filter. This would warrant a further experiment, starting with these values and trying larger filter sizes and perhaps fewer filters. I'll leave that to you to see if you can improve the model further!

Plugging these values into the model architecture, you'll get this:

```

dataset = windowed_dataset(x_train, window_size, batch_size,
                           shuffle_buffer_size)

model = tf.keras.models.Sequential([
    tf.keras.layers.Conv1D(filters=128, kernel_size=9,
                           strides=1, padding="causal",
                           activation="relu",
                           input_shape=[None, 1]),
    tf.keras.layers.Dense(28, input_shape=[window_size],
                           activation="relu"),
    tf.keras.layers.Dense(10, activation="relu"),
    tf.keras.layers.Dense(1),
])

```

```
optimizer = tf.keras.optimizers.SGD(lr=1e-5, momentum=0.5)
model.compile(loss="mse", optimizer=optimizer)
history = model.fit(dataset, epochs=100, verbose=1)
```

After training with this, the model had improved accuracy compared with both the naive CNN created earlier *and* the original DNN, giving [Figure 11-5](#).

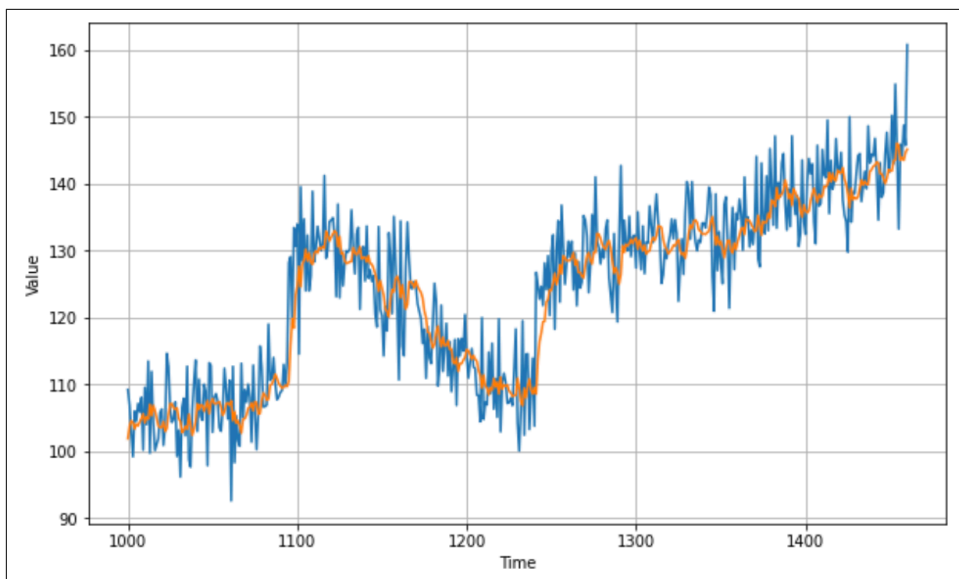


Figure 11-5. Optimized CNN predictions

This resulted in an MAE of 4.39, which is a slight improvement over the 4.47 we got without using the convolutional layer. Further experimentation with the CNN hyperparameters may improve this further.

Beyond convolutions, the techniques we explored in the chapters on natural language processing with RNNs, including LSTMs, may be powerful when working with sequence data. By their very nature, RNNs are designed for maintaining context, so previous values can have an effect on later ones. You'll explore using them for sequence modeling next. But first, let's move on from a synthetic dataset and start looking at real data. In this case, we'll consider weather data.

Using NASA Weather Data

One great resource for time series weather data is the [NASA Goddard Institute for Space Studies \(GISS\) Surface Temperature analysis](#). If you follow the [Station Data link](#), on the right side of the page you can pick a weather station to get data from. For example, I chose the Seattle Tacoma (SeaTac) airport and was taken to the page in [Figure 11-6](#).

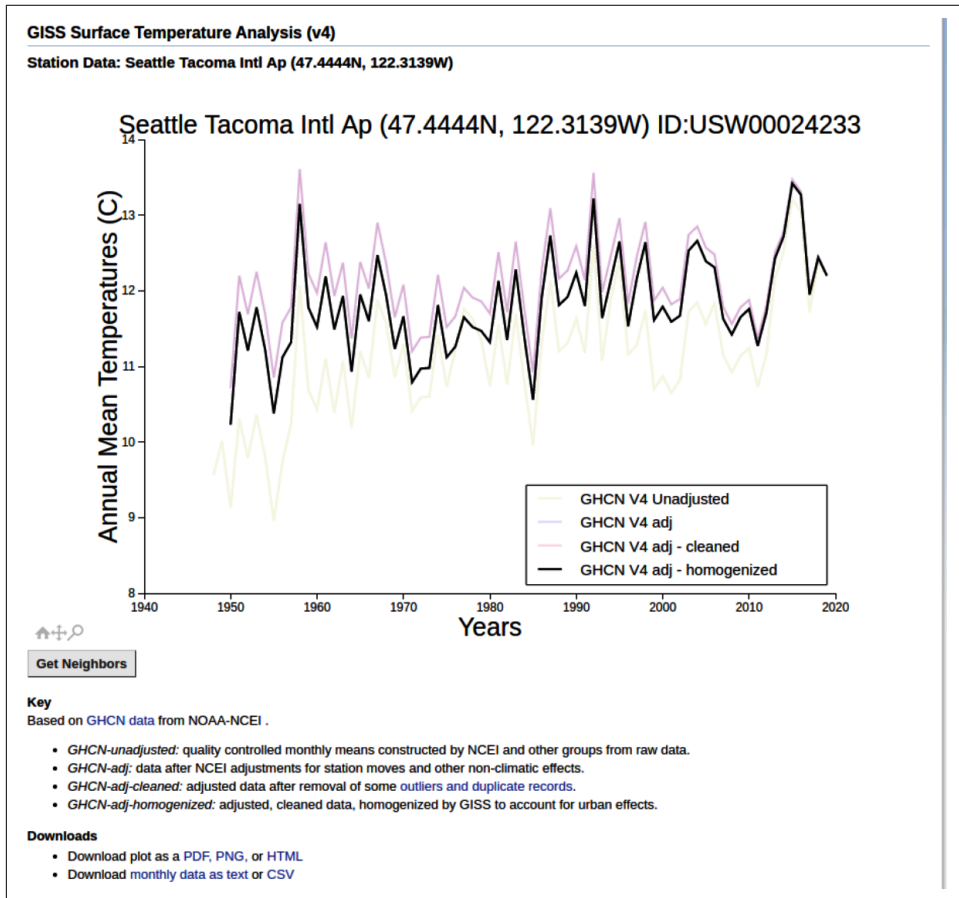


Figure 11-6. Surface temperature data from GISS

You can see a link to download monthly data as CSV at the bottom of this page. Select this, and a file called *station.csv* will be downloaded to your device. If you open this, you'll see that it's a grid of data with a year in each row and a month in each column, like in [Figure 11-7](#).

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	YEAR	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC
2	1950	-2.54	5.85	6.99	9.37	12.31	17.04	18.99	19.05	15.74	10.95	7.84	8.5
3	1951	4.14	6.19	5.49	11.15	13.73	17.57	19.38	17.86	16.43	11.85	8.38	3.87
4	1952	3.86	6.21	7.13	10.55	13.63	15.01	18.86	18.63	16.51	13.77	6.43	6.76
5	1953	8.25	5.95	7.49	9.63	12.85	14.45	18	18.49	16.77	13.17	9.56	7.12
6	1954	3.67	7.08	6.24	8.94	13.57	14.91	17.02	17.3	16.24	11.73	10.83	6.26
7	1955	5.38	4.86	5.37	8.39	11.78	15.83	17	17.49	15.3	11.62	5.23	4.96
8	1956	5.3	3.45	6.32	11.2	15.2	15.25	19.54	18.61	15.81	10.77	6.98	5.64

Figure 11-7. Exploring the data

As this is CSV data, it's pretty easy to process in Python, but as with any dataset, do note the format. When reading CSV, you tend to read it line by line, and often each line has one data point that you're interested in. In this case there are at least 12 data points of interest per line, so you'll have to consider this when reading the data.

Reading GISS Data in Python

The code to read the GISS data is shown here:

```
def get_data():
    data_file = "/home/ljpm/Desktop/bookpython/station.csv"
    f = open(data_file)
    data = f.read()
    f.close()
    lines = data.split('\n')
    header = lines[0].split(',')
    lines = lines[1:]
    temperatures=[]
    for line in lines:
        if line:
            linedata = line.split(',')
            linedata = linedata[1:13]
            for item in linedata:
                if item:
                    temperatures.append(float(item))

    series = np.asarray(temperatures)
    time = np.arange(len(temperatures), dtype="float32")
    return time, series
```

This will open the file at the indicated path (yours will of course differ) and read in the entire file as a set of lines, where the line split is the new line character (`\n`). It will then loop through each line, ignoring the first line, and split them on the comma character into a new array called `linedata`. The items from 1 through 13 in this array will indicate the values for the months January through February as strings. These values are converted to floats and added to the array called `temperatures`. Once it's completed it will be turned into a Numpy array called `series`, and another Numpy

array called `time` will be created that's the same size as `series`. As it is created using `np.arange`, the first element will be 1, the second 2, etc. Thus, this function will return `time` in steps from 1 to the number of data points, and `series` as the data for that time.

Now if you want a time series that is normalized, you can simply run this code:

```
time, series = get_data()
mean = series.mean(axis=0)
series -= mean
std = series.std(axis=0)
series /= std
```

This can be split into training and validation sets as before. Choose your split time based on the size of the data—in this case I had ~840 data items, so I split at 792 (reserving four years' worth of data points for validation):

```
split_time = 792
time_train = time[:split_time]
x_train = series[:split_time]
time_valid = time[split_time:]
x_valid = series[split_time:]
```

Because the data is now a Numpy array, you can use the same code as before to create a windowed dataset from it to train a neural network:

```
window_size = 24
batch_size = 12
shuffle_buffer_size = 48
dataset = windowed_dataset(x_train, window_size,
                           batch_size, shuffle_buffer_size)
valid_dataset = windowed_dataset(x_valid, window_size,
                                 batch_size, shuffle_buffer_size)
```

This should use the same `windowed_dataset` function as the convolutional network earlier in this chapter, adding a new dimension. When using RNNs, GRUs, and LSTMs, you will need the data in that shape.

Using RNNs for Sequence Modeling

Now that you have the data from the NASA CSV in a windowed dataset, it's relatively easy to create a model to train a predictor for it. (It's a bit more difficult to train a *good* one!) Let's start with a simple, naive model using RNNs. Here's the code:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.SimpleRNN(100, return_sequences=True,
                              input_shape=[None, 1]),
    tf.keras.layers.SimpleRNN(100),
    tf.keras.layers.Dense(1)
])
```

In this case, the Keras `SimpleRNN` layer is used. RNNs are a class of neural networks that are powerful for exploring sequence models. You first saw them in [Chapter 7](#) when you were looking at natural language processing. I won't go into detail on how they work here, but if you're interested and you skipped that chapter, take a look back at it now. Notably, an RNN has an internal loop that iterates over the time steps of a sequence while maintaining an internal state of the time steps it has seen so far. A `SimpleRNN` has the output of each time step fed into the next time step.

You can compile and fit the model with the same hyperparameters as before, or use Keras Tuner to see if you can find better ones. For simplicity, you can use these settings:

```
optimizer = tf.keras.optimizers.SGD(lr=1.5e-6, momentum=0.9)
model.compile(loss=tf.keras.losses.Huber(),
              optimizer=optimizer, metrics=["mae"])

history = model.fit(dataset, epochs=100, verbose=1,
                    validation_data=valid_dataset)
```

Even one hundred epochs is enough to get an idea of how it can predict values. [Figure 11-8](#) shows the results.

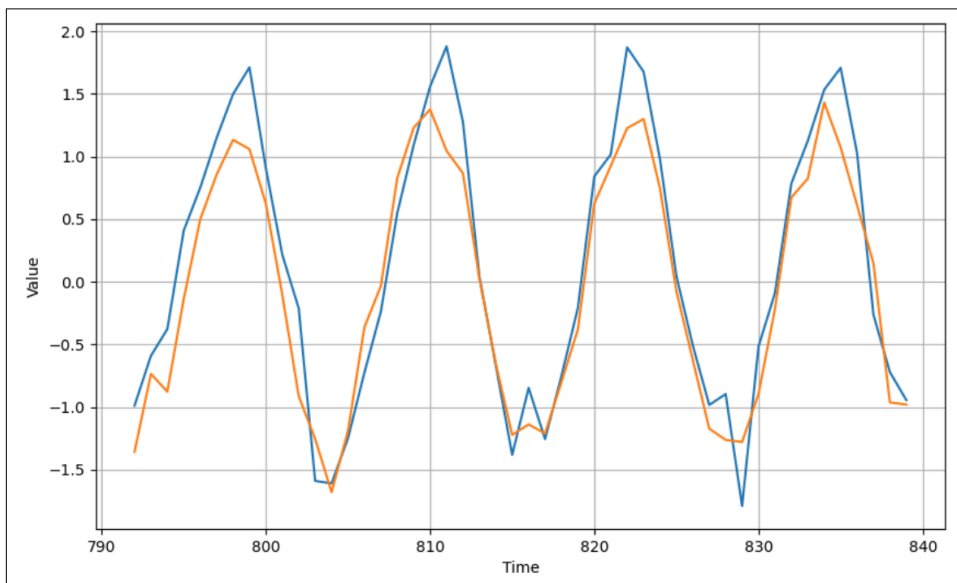


Figure 11-8. Results of the SimpleRNN

As you can see, the results were pretty good. It may be a little off in the peaks, and when the pattern changes unexpectedly (like at time steps 815 and 828), but on the whole it's not bad. Now let's see what happens if we train it for 1,500 epochs ([Figure 11-9](#)).

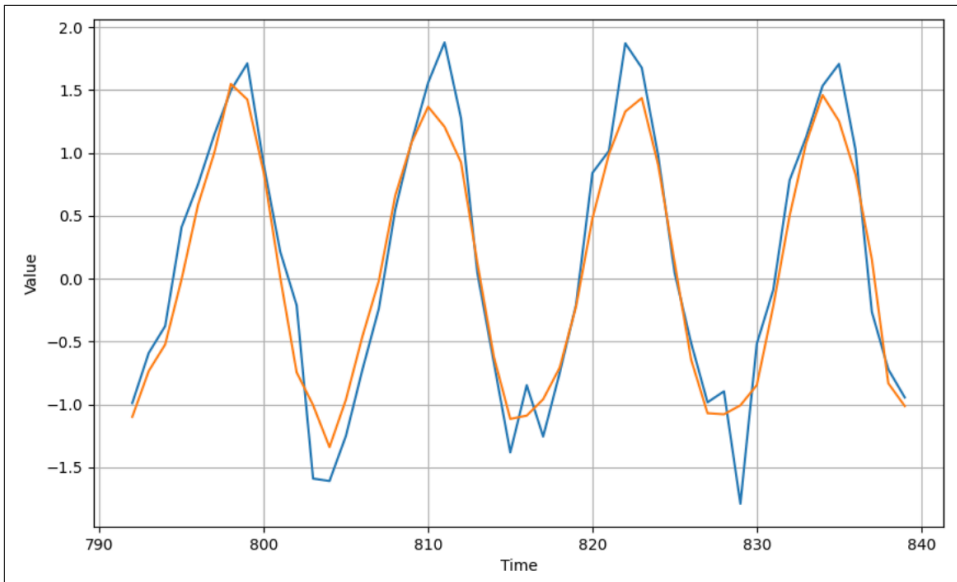


Figure 11-9. RNN trained over 1,500 epochs

There's not much of a difference, except that some of the peaks are smoothed out. If you look at the history of loss on both the validation and training sets, it looks like [Figure 11-10](#).

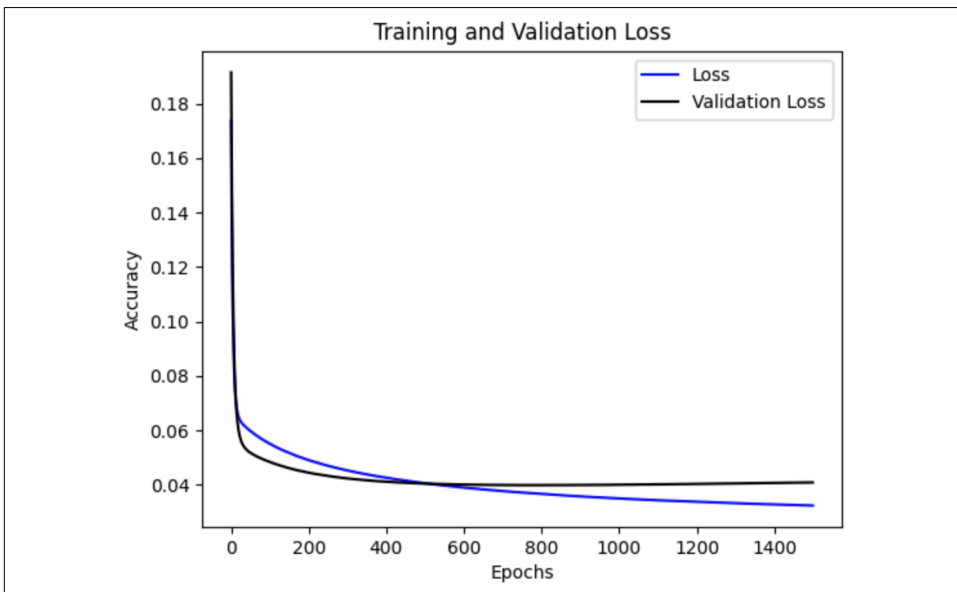


Figure 11-10. Training and validation loss for the SimpleRNN

As you can see, there's a healthy match between the training loss and the validation loss, but as the epochs increase, the model begins to overfit on the training set. Perhaps a better number of epochs would be around five hundred.

One reason for this could be the fact that the data, being monthly weather data, is highly seasonal. Another is that there is a very large training set and a relatively small validation set. Next, we'll explore using a larger climate dataset.

Exploring a Larger Dataset

The **KNMI Climate Explorer** allows you to explore granular climate data from many locations around the world. I downloaded a **dataset** consisting of daily temperature readings from the center of England from 1772 until 2020. This data is structured differently from the GISS data, with the date as a string, followed by a number of spaces, followed by the reading.

I've prepared the data, stripping the headers and removing the extraneous spaces. That way it's easy to read with code like this:

```
def get_data():
    data_file = "tdaily_cet.dat.txt"
    f = open(data_file)
    data = f.read()
    f.close()
    lines = data.split('\n')
    temperatures=[]
    for line in lines:
        if line:
            linedata = line.split(' ')
            temperatures.append(float(linedata[1]))

    series = np.asarray(temperatures)
    time = np.arange(len(temperatures), dtype="float32")
    return time, series
```

This dataset has 90,663 data points in it, so, before training your model, be sure to split it appropriately. I used a split time of 80,000, leaving 10,663 records for validation. Also, update the window size, batch size, and shuffle buffer size appropriately. Here's an example:

```
window_size = 60
batch_size = 120
shuffle_buffer_size = 240
```

Everything else can remain the same. As you can see in **Figure 11-11**, after training for one hundred epochs, the plot of the predictions against the validation set looks pretty good.

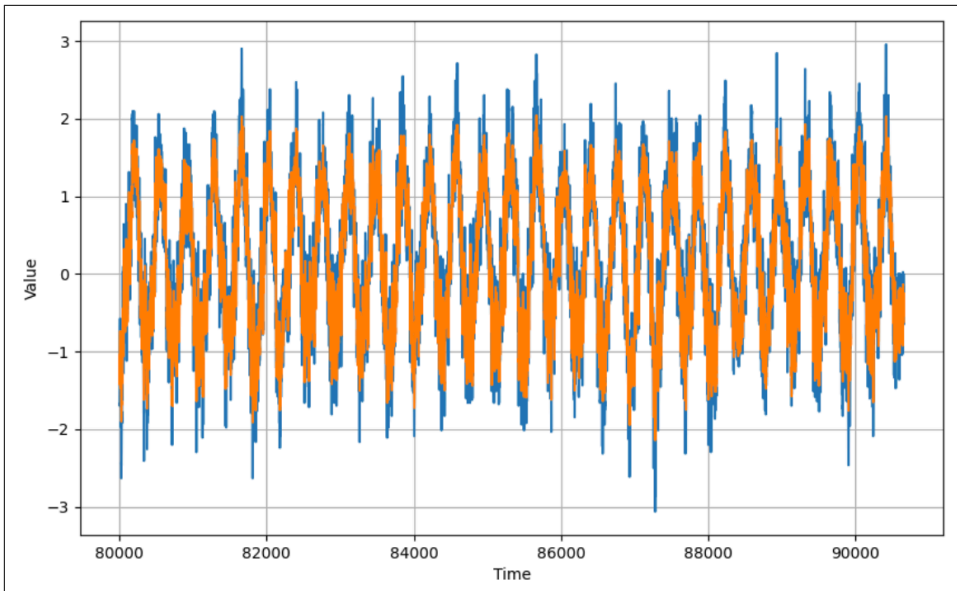


Figure 11-11. Plot of predictions against real data

There's a lot of data here, so let's zoom in to the last hundred days' worth (Figure 11-12).

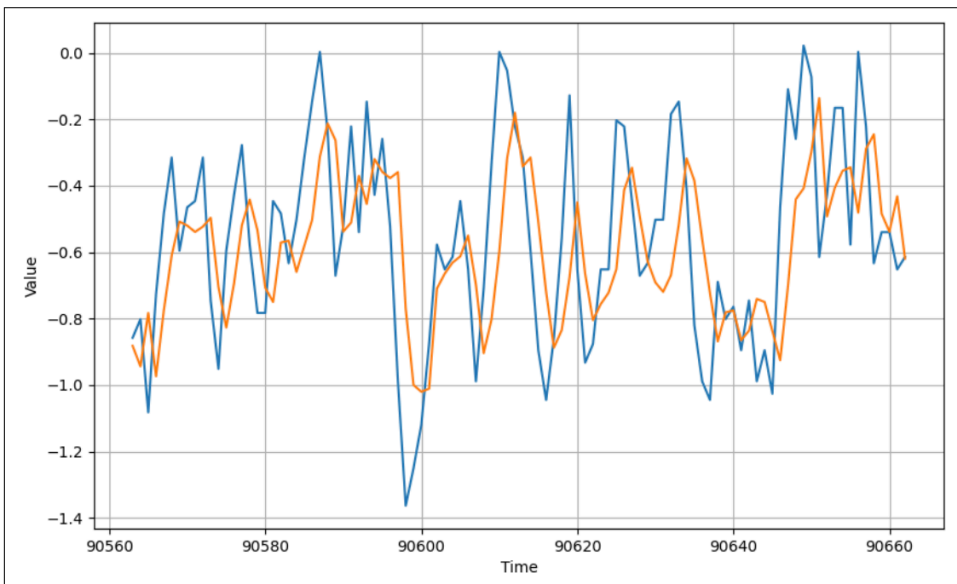


Figure 11-12. Results for one hundred days' worth of data

While the chart generally follows the curve of the data, and is getting the trends roughly correct, it is pretty far off, particularly at the extreme ends, so there's room for improvement.

It's also important to remember that we normalized the data, so while our loss and MAE may look low, that's because they are based on the loss and MAE of normalized values that have a much lower variance than the real ones. So, [Figure 11-13](#), showing loss of less than 0.1, might lead you into a false sense of security.

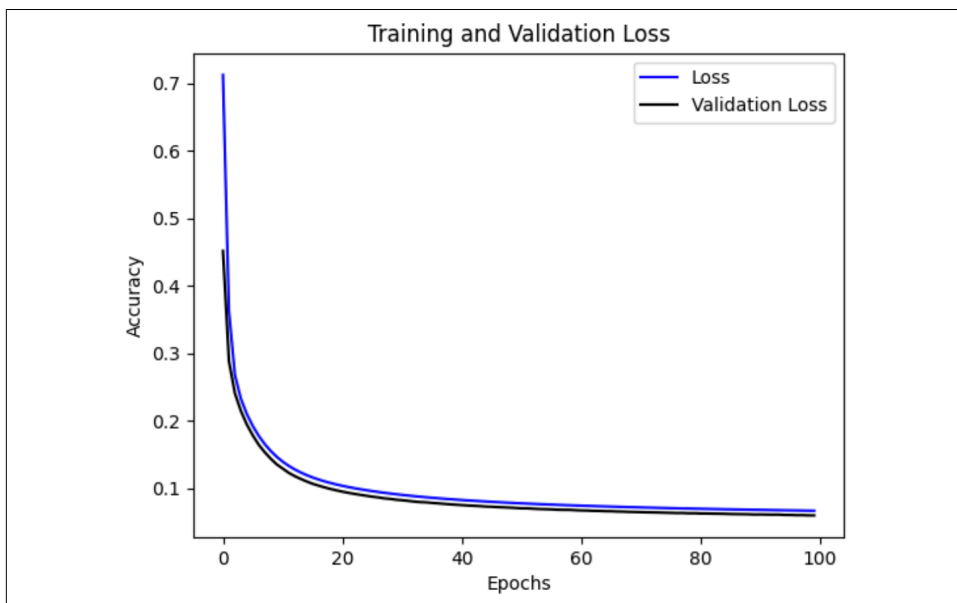


Figure 11-13. Loss and validation loss for large dataset

To denormalize the data, you can do the inverse of normalization: first multiply by the standard deviation, and then add back the mean. At that point, if you wish, you can calculate the real MAE for the prediction set as done previously.

Using Other Recurrent Methods

In addition to the SimpleRNN, TensorFlow has other recurrent layer types, such as gated recurrent units (GRUs) and long short-term memory layers (LSTMs), discussed in [Chapter 7](#). When using the TFRecord-based architecture for your data that you've been using throughout this chapter, it becomes relatively simple to just drop in these RNN types if you want to experiment.

So, for example, if you consider the simple naive RNN that you created earlier:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.SimpleRNN(100, input_shape=[None, 1],
                               return_sequences=True),
    tf.keras.layers.SimpleRNN(100),
    tf.keras.layers.Dense(1)
])
```

Replacing this with a GRU becomes as easy as:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.GRU(100, input_shape=[None, 1], return_sequences=True),
    tf.keras.layers.GRU(100),
    tf.keras.layers.Dense(1)
])
```

With an LSTM, it's similar:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(100, input_shape=[None, 1], return_sequences=True),
    tf.keras.layers.LSTM(100),
    tf.keras.layers.Dense(1)
])
```

It's worth experimenting with these layer types as well as with different hyperparameters, loss functions, and optimizers. There's no one-size-fits-all solution, so what works best for you in any given situation will depend on your data and your requirements for prediction with that data.

Using Dropout

If you encounter overfitting in your models, where the MAE or loss for the training data is much better than with the validation data, you can use dropout. As discussed in [Chapter 3](#) in the context of computer vision, with dropout, neighboring neurons are randomly dropped out (ignored) during training to avoid a familiarity bias. When using RNNs, there's also a *recurrent dropout* parameter that you can use.

What's the difference? Recall that when using RNNs you typically have an input value, and the neuron calculates an output value and a value that gets passed to the next time step. Dropout will randomly drop out the input values. Recurrent dropout will randomly drop out the recurrent values that get passed to the next step.

For example, consider the basic recurrent neural network architecture shown in [Figure 11-14](#).

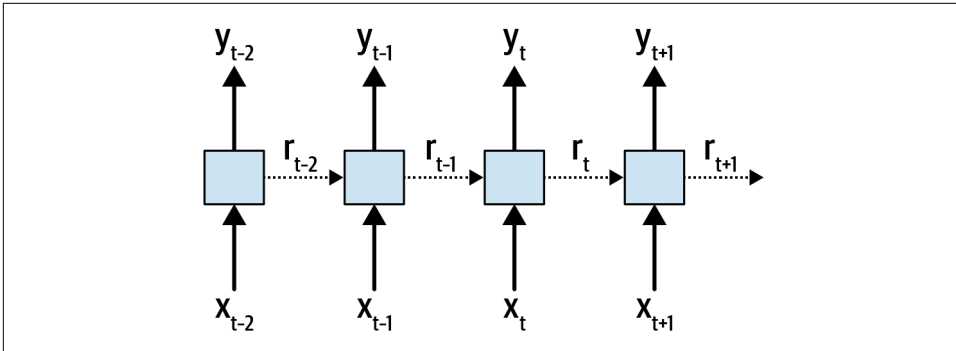


Figure 11-14. Recurrent neural network

Here you can see the inputs to the layers at different time steps (x). The current time is t , and the steps shown are $t - 2$ through $t + 1$. The relevant outputs at the same time steps (y) are also shown. The recurrent values passed between time steps are indicated by the dotted lines and labeled as r .

Using *dropout* will randomly drop out the x inputs. Using *recurrent dropout* will randomly drop out the r recurrent values.

You can learn more about how recurrent dropout works from a deeper mathematical perspective in the paper “[A Theoretically Grounded Application of Dropout in Recurrent Neural Networks](#)” by Yarin Gal and Zoubin Ghahramani.

One thing to consider when using recurrent dropout is discussed by Gal in his research around [uncertainty in deep learning](#), where he demonstrates that the same pattern of dropout units should be applied at every time step, and that a similar constant dropout mask should be applied at every time step. While dropout is typically random, Gal’s work was built into Keras, so when using `tf.keras` the consistency recommended by his research is maintained.

To add dropout and recurrent dropout, you simply use the relevant parameters on your layers. For example, adding them to the simple GRU from earlier would look like this:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.GRU(100, input_shape=[None, 1], return_sequences=True,
                        dropout=0.1, recurrent_dropout=0.1),
    tf.keras.layers.GRU(100, dropout=0.1, recurrent_dropout=0.1),
    tf.keras.layers.Dense(1),
])
```

Each parameter takes a value between 0 and 1 indicating the proportion of values to drop out. A value of 0.1 will drop out 10% of the requisite values.

RNNs using dropout will often take longer to converge, so be sure to train them for more epochs to test for this. [Figure 11-15](#) shows the results of training the preceding GRU with dropout and recurrent dropout on each layer set to 0.1 over 1,000 epochs.

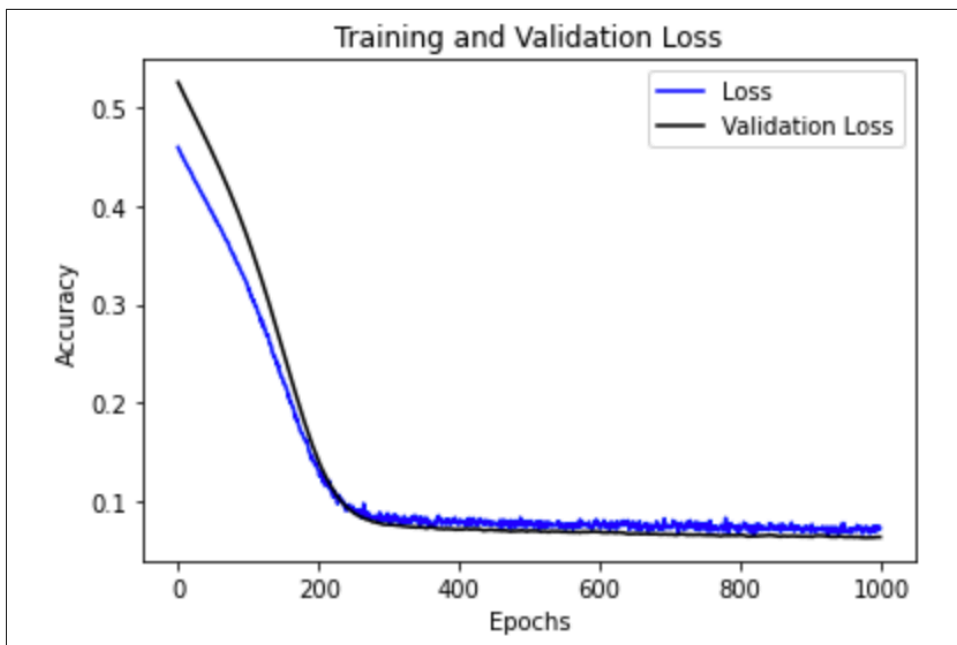


Figure 11-15. Training a GRU with dropout

As you can see, the loss and MAE decreased rapidly until about epoch 300, after which they continued to decline, but quite noisily. You'll often see noise like this in the loss when using dropout, and it's an indication that you may want to tweak the amount of dropout as well as the parameters of the loss function, such as learning rate. Predictions with this network were shaped quite nicely, as you can see in [Figure 11-16](#), but there's room for improvement, in that the peaks of the predictions are much lower than the real peaks.

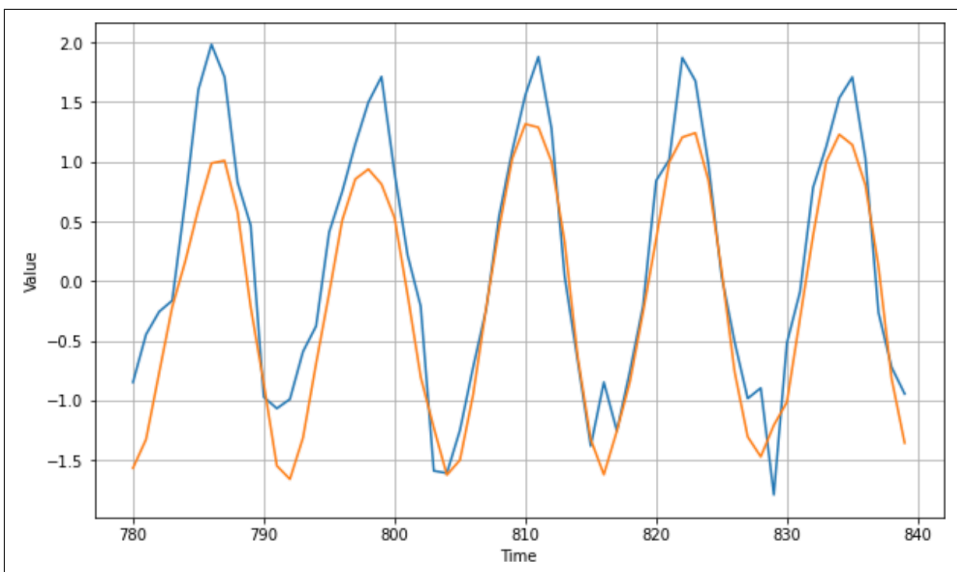


Figure 11-16. Predictions using a GRU with dropout

As you’ve seen in this chapter, predicting time sequence data using neural networks is a difficult proposition, but tweaking their hyperparameters (particularly with tools such as Keras Tuner) can be a powerful way to improve your model and its subsequent predictions.

Using Bidirectional RNNs

Another technique to consider when classifying sequences is to use bidirectional training. This may seem counterintuitive at first, as you might wonder how future values could impact past ones. But recall that time series values can contain seasonality, where values repeat over time, and when using a neural network to make predictions all we’re doing is sophisticated pattern matching. Given that data repeats, a signal for how data can repeat might be found in future values—and when using bidirectional training, we can train the network to try to spot patterns going from time t to time $t + x$, as well as going from time $t + x$ to time t .

Fortunately, coding this is simple. For example, consider the GRU from the previous section. To make this bidirectional, you simply wrap each GRU layer in a `tf.keras.layers.Bidirectional` call. This will effectively train twice on each step—once with the sequence data in the original order, and once with it in reverse order. The results are then merged before proceeding to the next step.

Here's an example:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Bidirectional(
        tf.keras.layers.GRU(100, input_shape=[None, 1], return_sequences=True,
                               dropout=0.1, recurrent_dropout=0.1)),
    tf.keras.layers.Bidirectional(
        tf.keras.layers.GRU(100, dropout=0.1, recurrent_dropout=0.1)),
    tf.keras.layers.Dense(1),
])
```

A plot of the results of training with a bidirectional GRU with dropout on the time series is shown in [Figure 11-17](#). As you can see, there's no major difference here, and the MAE ends up being similar. However, with a larger data series, you may see a decent accuracy difference, and additionally tweaking the training parameters—particularly `window_size`, to get multiple seasons—can have a pretty big impact.

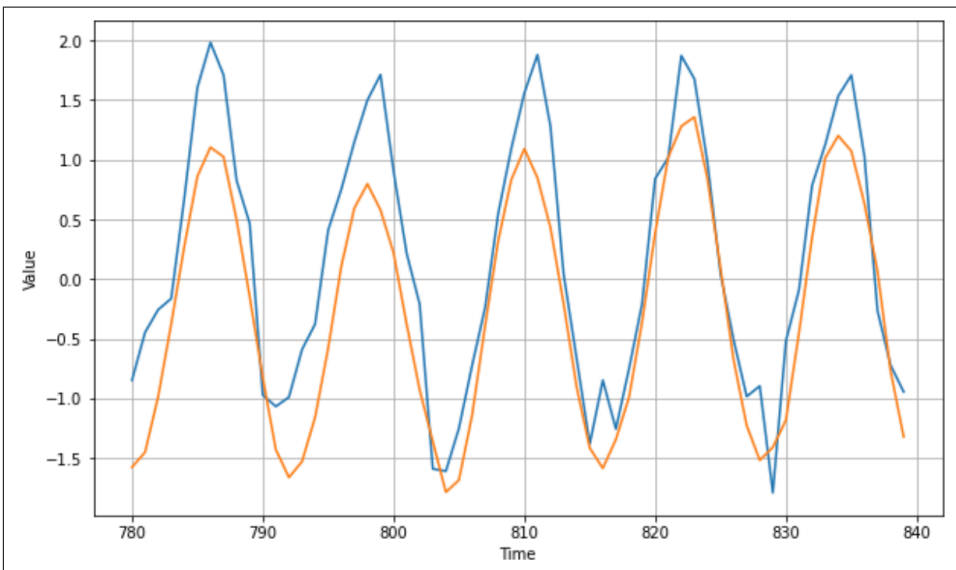


Figure 11-17. Training with a bidirectional GRU

This network has an MAE (on the normalized data) of about .48, chiefly because it doesn't seem to do too well on the high peaks. Retraining it with a larger window and bidirectionality produces better results: it has a significantly lower MAE of about .28 ([Figure 11-18](#)).

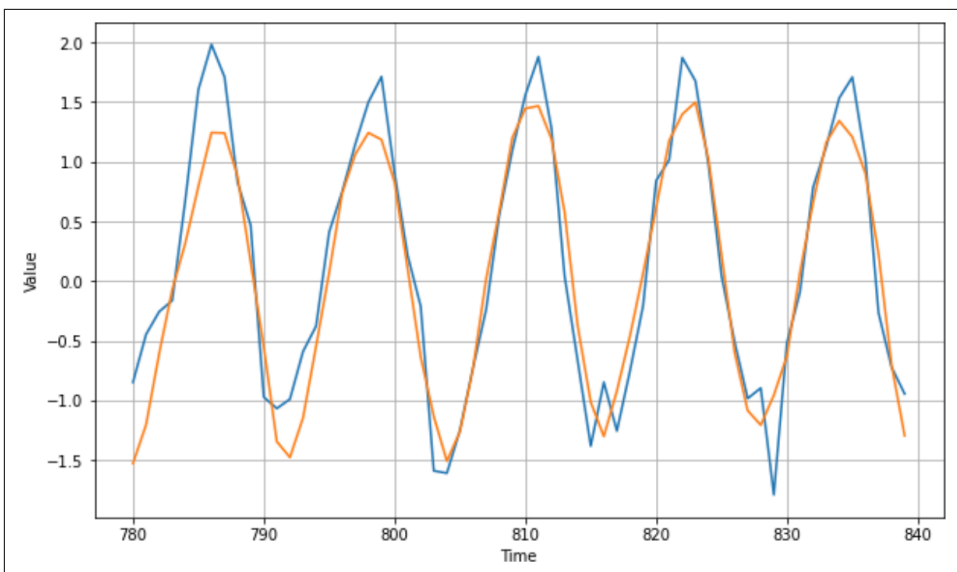


Figure 11-18. Larger window, bidirectional GRU results

As you can see, you can experiment with different network architectures and different hyperparameters to improve your overall predictions. The ideal choices are very much dependent on the data, so the skills you’ve learned in this chapter will help you with your specific datasets!

Summary

In this chapter, you explored different network types for building models to predict time series data. You built on the simple DNN from [Chapter 10](#), adding convolutions, and experimented with recurrent network types such as simple RNNs, GRUs, and LSTMs. You saw how you can tweak hyperparameters and the network architecture to improve your model’s accuracy, and you practiced working with some real-world datasets, including one massive dataset with hundreds of years’ worth of temperature readings. You’re now ready to get started building networks for a variety of datasets, with a good understanding of what you need to know to optimize them!

PART II

Using Models

An Introduction to TensorFlow Lite

In all of the chapters of this book so far, you’ve been exploring how to use TensorFlow to create machine learning models that can provide functionality such as computer vision, natural language processing, and sequence modeling without the explicit programming of rules. Instead, using labeled data, neural networks are able to learn the patterns that distinguish one thing from another, and this can then be extended into solving problems. For the rest of the book we’re going to switch gears and look at *how* to use these models in common scenarios. The first, most obvious and perhaps most useful topic we’ll cover is how to use models in mobile applications. In this chapter, I’ll go over the underlying technology that makes it possible to do machine learning on mobile (and embedded) devices: TensorFlow Lite. Then, in the next two chapters we’ll explore scenarios of using these models on Android and iOS.

TensorFlow Lite is a suite of tools that complements TensorFlow, achieving two main goals. The first is to make your models mobile-friendly. This often involves reducing their size and complexity, with as little impact as possible on their accuracy, to make them work better in a battery-constrained environment like a mobile device. The second is to provide a runtime for different mobile platforms, including Android, iOS, mobile Linux (for example, Raspberry Pi), and various microcontrollers. Note that you cannot *train* a model with TensorFlow Lite. Your workflow will be to train it using TensorFlow and then *convert* it to the TensorFlow Lite format, before loading and running it using a TensorFlow Lite interpreter.

What Is TensorFlow Lite?

TensorFlow Lite started as a mobile version of TensorFlow aimed at Android and iOS developers, with the goal of being an effective ML toolkit for their needs. When building and executing models on computers or cloud services, issues like battery consumption, screen size, and other aspects of mobile app development aren’t a

concern, so when mobile devices are targeted a new set of constraints need to be addressed.

The first is that a mobile application framework needs to be *lightweight*. Mobile devices have far more limited resources than the typical machine that is used for training models. As such, developers have to be very careful about the resources that are used not just by the application, but also the application framework. Indeed, when users are browsing an app store they'll see the size of each application and have to make a decision about whether to download it based on their data usage. If the framework that runs a model is large, and the model itself is also large, this will bloat the file size and turn the user off.

The framework also has to be *low latency*. Apps that run on mobile devices need to perform well, or the user may stop using them. Only 38% of apps are used more than 11 times, meaning 62% of apps are used 10 times or less. Indeed, 25% of all apps are only used once. High latency, where an app is slow to launch or process key data, is a factor in this abandonment rate. Thus, a framework for ML-based apps needs to be fast to load and fast to perform the necessary inference.

In partnership with low latency, a mobile framework needs an *efficient model format*. When training on a powerful supercomputer, the model format generally isn't the most important signal. As we've seen in earlier chapters, high model accuracy, low loss, avoiding overfitting, etc. are the metrics a model creator will chase. But when running on a mobile device, in order to be lightweight and have low latency, the model format will need to be taken into consideration as well. Much of the math in the neural networks we've seen so far is floating-point operations with high precision. For scientific discovery, that's essential. For running on a mobile device it may not be. A mobile-friendly framework will need to help you with trade-offs like this and give you the tools to convert your model if necessary.

Having your models run on-device has a major benefit in that they don't need to pass data to a cloud service to have inference performed on them. This leads to improvements in *user privacy* as well as *power consumption*. Not needing to use the radio for a cellular or WiFi signal to send the data and receive the predictions is good, so long as the on-device inference doesn't cost more power-wise. Keeping data on the device in order to run predictions is also a powerful and increasingly important feature, for obvious reasons! (Later in this book we'll discuss *federated learning*, which is a hybrid of on-device and cloud-based machine learning that gives you the best of both worlds, while also maintaining privacy.)

So, with all of this in mind, TensorFlow Lite was created. As mentioned earlier, it's not a framework for *training* models, but a supplementary set of tools designed to meet all the constraints of mobile and embedded systems.

It should broadly be seen as two main things: a converter that takes your TensorFlow model and converts it to the *.tflite* format, shrinking and optimizing it, and a suite of interpreters for various runtimes ([Figure 12-1](#)).

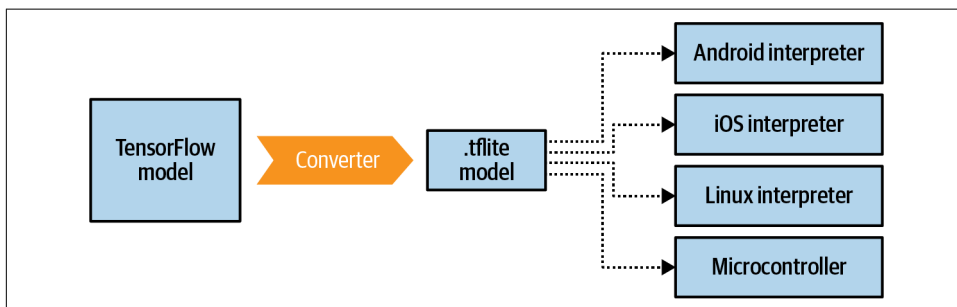


Figure 12-1. The TensorFlow Lite suite

The interpreter environments also support acceleration options within their particular frameworks. For example, on Android the [Neural Networks API](#) is supported, so TensorFlow Lite can take advantage of it on devices where it's available.

Note that not every operation (or “op”) in TensorFlow is presently supported in TensorFlow Lite or the TensorFlow Lite converter. You may encounter this issue when converting models, and it's always a good idea to check the [documentation](#) for details. One helpful piece of workflow, as you'll see later in this chapter, is to take an existing mobile-friendly model and use transfer learning for your scenario. You can find lists of models optimized to work with TensorFlow Lite on the [TensorFlow website](#) and [TensorFlow Hub](#).

Walkthrough: Creating and Converting a Model to TensorFlow Lite

We'll begin with a step-by-step walkthrough showing how to create a simple model with TensorFlow, convert it to the TensorFlow Lite format, and then use the TensorFlow Lite interpreter. For this walkthrough I'll use the Linux interpreter because it's readily available in Google Colab. In [Chapter 13](#) you'll see how to use this model on Android, and in [Chapter 14](#) you'll explore using it on iOS.

Back in [Chapter 1](#) you saw a very simple TensorFlow model that learned the relationship between two sets of numbers that ended up as $Y = 2X - 1$. For convenience, here's the complete code:


```

l0 = Dense(units=1, input_shape=[1])
model = Sequential([l0])
model.compile(optimizer='sgd', loss='mean_squared_error')

xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

model.fit(xs, ys, epochs=500)

print(model.predict([10.0]))
print("Here is what I learned: {}".format(l0.get_weights()))

```

Once this has been trained, as you saw, you can do `model.predict[x]` and get the expected `y`. In the preceding code, `x=10`, and the `y` the model will give us back is a value close to 19.

As this model is small and easy to train, we can use it as an example that we'll convert to TensorFlow Lite to show all the steps.

Step 1. Save the Model

The TensorFlow Lite converter works on a number of different file formats, including SavedModel (preferred) and the Keras H5 format. For this exercise we'll use SavedModel.

To do this, simply specify a directory in which to save the model and call `tf.saved_model.save`, passing it the model and directory:

```

export_dir = 'saved_model/1'
tf.saved_model.save(model, export_dir)

```

The model will be saved out as assets and variables as well as a *saved_model.pb* file, as shown in [Figure 12-2](#).

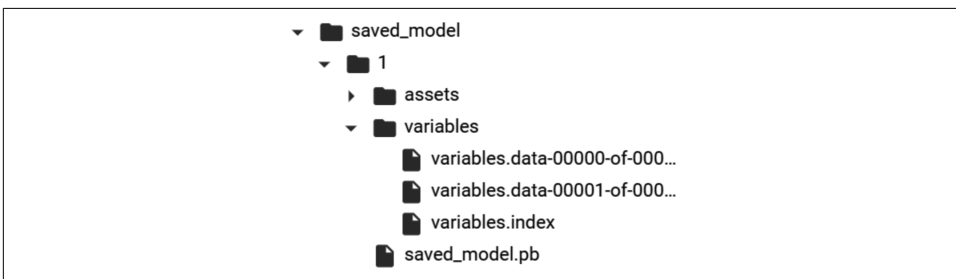


Figure 12-2. SavedModel structure

Once you have the saved model, you can convert it using the TensorFlow Lite converter.



The TensorFlow team recommends using the SavedModel format for compatibility across the entire TensorFlow ecosystem, including future compatibility with new APIs.

Step 2. Convert and Save the Model

The TensorFlow Lite converter is in the `tf.lite` package. You can call it to convert a saved model by first invoking it with the `from_saved_model` method, passing it the directory containing the saved model, and then invoking its `convert` method:

```
# Convert the model.
converter = tf.lite.TFLiteConverter.from_saved_model(export_dir)
tflite_model = converter.convert()
```

You can then save out the new *.tflite* model using `pathlib`:

```
import pathlib
tflite_model_file = pathlib.Path('model.tflite')
tflite_model_file.write_bytes(tflite_model)
```

At this point, you have a *.tflite* file that you can use in any of the interpreter environments. Later we'll use it on Android and iOS, but, for now, let's use the Python-based interpreter so you can run it in Colab. This same interpreter can be used in embedded Linux environments like a Raspberry Pi!

Step 3. Load the TFLite Model and Allocate Tensors

The next step is to load the model into the interpreter, allocate tensors that will be used for inputting data to the model for prediction, and then read the predictions that the model outputs. This is where using TensorFlow Lite, from a programmer's perspective, greatly differs from using TensorFlow. With TensorFlow you can just say `model.predict(something)` and get the results, but because TensorFlow Lite won't have many of the dependencies that TensorFlow does, particularly in non-Python environments, you now have to get a bit more low-level and deal with the input and output tensors, formatting your data to fit them and parsing the output in a way that makes sense for your device.

First, load the model and allocate the tensors:

```
interpreter = tf.lite.Interpreter(model_content=tflite_model)
interpreter.allocate_tensors()
```

Then you can get the input and output details from the model, so you can begin to understand what data format it expects, and what data format it will provide back to you:

```
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
```

```
print(input_details)
print(output_details)
```

You'll get a lot of output!

First, let's inspect the input parameter. Note the shape setting, which is an array of type [1,1]. Also note the class, which is `numpy.float32`. These settings will dictate the shape of the input data and its format:

```
{'name': 'dense_input', 'index': 0, 'shape': array([1, 1], dtype=int32),
 'shape_signature': array([1, 1], dtype=int32), 'dtype': <class
 'numpy.float32'>, 'quantization': (0.0, 0), 'quantization_parameters':
 {'scales': array([], dtype=float32), 'zero_points': array([], dtype=int32),
 'quantized_dimension': 0}, 'sparsity_parameters': {}}
```

So, in order to format the input data, you'll need to use code like this to define the input array shape and type if you want to predict the y for $x=10.0$:

```
to_predict = np.array([[10.0]], dtype=np.float32)
print(to_predict)
```

The double brackets around the 10.0 can cause a little confusion—the mnemonic I use for the `array[1,1]` here is to say that there is 1 list, giving us the first set of [], and that list contains just 1 value, which is [10.0], thus giving [[10.0]]. It can also be confusing that the shape is defined as `dtype=int32`, whereas you're using `numpy.float32`. The `dtype` parameter is the data type defining the shape, not the contents of the list that is encapsulated in that shape. For that, you'll use the class.

The output details are very similar, and what you want to keep an eye on here is the shape. Because it's also an array of type [1,1], you can expect the answer to be [[y]] in much the same way as the input was [[x]]:

```
{'name': 'Identity', 'index': 3, 'shape': array([1, 1], dtype=int32),
 'shape_signature': array([1, 1], dtype=int32), 'dtype': <class
 'numpy.float32'>, 'quantization': (0.0, 0), 'quantization_parameters':
 {'scales': array([], dtype=float32), 'zero_points': array([], dtype=int32),
 'quantized_dimension': 0}, 'sparsity_parameters': {}}
```

Step 4. Perform the Prediction

To get the interpreter to do the prediction, you set the input tensor with the value to predict, telling it what input value to use:

```
interpreter.set_tensor(input_details[0]['index'], to_predict)
interpreter.invoke()
```

The input tensor is specified using the index of the array of input details. In this case you have a very simple model that has only a single input option, so it's `input_details[0]`, and you'll address it at the index. Input details item 0 has only one index, indexed at 0, and it expects a shape of [1,1] as defined earlier. So, you put

the `to_predict` value in there. Then you invoke the interpreter with the `invoke` method.

You can then read the prediction by calling `get_tensor` and supplying it with the details of the tensor you want to read:

```
tflite_results = interpreter.get_tensor(output_details[0]['index'])
print(tflite_results)
```

Again, there's only one output tensor, so it will be `output_details[0]`, and you specify the index to get the details beneath it, which will have the output value.

So, for example, if you run this code:

```
to_predict = np.array([[10.0]], dtype=np.float32)
print(to_predict)
interpreter.set_tensor(input_details[0]['index'], to_predict)
interpreter.invoke()
tflite_results = interpreter.get_tensor(output_details[0]['index'])
print(tflite_results)
```

you should see output like:

```
[[10.]]
[[18.975412]]
```

where 10 is the input value and 18.97 is the predicted value, which is very close to 19, which is $2X - 1$ when $X = 10$. For why it's not 19, look back to [Chapter 1](#)!

Given that this is a very simple example, let's look at something a little more complex next—using transfer learning on a well-known image classification model, and then converting that for TensorFlow Lite. From there we'll also be able to better explore the impacts of optimizing and quantizing the model.

The Standalone Interpreter

TensorFlow Lite is part of the overall TensorFlow ecosystem and consists of the tools needed for converting trained models and interpreting models. In the next two chapters you'll see how to use the interpreters for Android and iOS, respectively, but there's also a [standalone Python-based interpreter](#) that can be installed on any system that runs Python, such as a Raspberry Pi. This gives you a way to run your models on embedded systems that can run Python. Syntactically, it works in the same way as the interpreter you just saw.

Walkthrough: Transfer Learning an Image Classifier and Converting to TensorFlow Lite

In this section we'll build a new version of the Dogs vs. Cats computer vision model from Chapters 3 and 4 that uses transfer learning. This will use a model from TensorFlow Hub, so if you need to install it, you can follow the [instructions on the site](#).

Step 1. Build and Save the Model

First, get all of the data:

```
import numpy as np
import matplotlib.pyplot as plt

import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_datasets as tfds

def format_image(image, label):
    image = tf.image.resize(image, IMAGE_SIZE) / 255.0
    return image, label

(raw_train, raw_validation, raw_test), metadata = tfds.load(
    'cats_vs_dogs',
    split=['train[:80%]', 'train[80%:90%]', 'train[90%:]'],
    with_info=True,
    as_supervised=True,
)

num_examples = metadata.splits['train'].num_examples
num_classes = metadata.features['label'].num_classes
print(num_examples)
print(num_classes)

BATCH_SIZE = 32
train_batches =
raw_train.shuffle(num_examples // 4)
.map(format_image).batch(BATCH_SIZE).prefetch(1)

validation_batches = raw_validation.map(format_image)
.batch(BATCH_SIZE).prefetch(1)
test_batches = raw_test.map(format_image).batch(1)
```

This will download the Dogs vs. Cats dataset and split it into training, test, and validation sets.

Next, you'll use the `mobilenet_v2` model from TensorFlow Hub to create a Keras layer called `feature_extractor`:

```

module_selection = ("mobilenet_v2", 224, 1280)
handle_base, pixels, FV_SIZE = module_selection

MODULE_HANDLE = "https://tfhub.dev/google/tf2-preview/{}/feature_vector/4"
.format(handle_base)

IMAGE_SIZE = (pixels, pixels)

feature_extractor = hub.KerasLayer(MODULE_HANDLE,
                                   input_shape=IMAGE_SIZE + (3,),
                                   output_shape=[FV_SIZE],
                                   trainable=False)

```

Now that you have the feature extractor, you can make it the first layer in your neural network and add an output layer with as many neurons as you have classes (in this case, two). You can then compile and train it:

```

model = tf.keras.Sequential([
    feature_extractor,
    tf.keras.layers.Dense(num_classes, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

hist = model.fit(train_batches,
                 epochs=5,
                 validation_data=validation_batches)

```

With just five epochs of training, this should give a model with 99% accuracy on the training set and 98%+ on the validation set. Now you can simply save the model out:

```

CATS_VS_DOGS_SAVED_MODEL = "exp_saved_model"
tf.saved_model.save(model, CATS_VS_DOGS_SAVED_MODEL)

```

Once you have the saved model, you can convert it.

Step 2. Convert the Model to TensorFlow Lite

As before, you can now take the saved model and convert it into a *.tflite* model. You'll save it out as *converted_model.tflite*:

```

converter = tf.lite.TFLiteConverter.from_saved_model(CATS_VS_DOGS_SAVED_MODEL)
tflite_model = converter.convert()
tflite_model_file = 'converted_model.tflite'

with open(tflite_model_file, "wb") as f:
    f.write(tflite_model)

```

Once you have the file, you can instantiate an interpreter with it. When this is done, you should get the input and output details as before. Load them into variables called

input_index and output_index, respectively. This makes the code a little more readable!

```
interpreter = tf.lite.Interpreter(model_path=tlite_model_file)
interpreter.allocate_tensors()

input_index = interpreter.get_input_details()[0]["index"]
output_index = interpreter.get_output_details()[0]["index"]

predictions = []
```

The dataset has lots of test images in test_batches, so if you want to take one hundred of the images and test them, you can do so like this (feel free to change the 100 to any other value):

```
test_labels, test_imgs = [], []
for img, label in test_batches.take(100):
    interpreter.set_tensor(input_index, img)
    interpreter.invoke()
    predictions.append(interpreter.get_tensor(output_index))
    test_labels.append(label.numpy()[0])
    test_imgs.append(img)
```

Earlier, when reading the images, they were reformatted by the mapping function called format_image to be the right size for both training and inference, so all you have to do now is set the interpreter's tensor at the input index to the image. After invoking the interpreter, you can then get the tensor at the output index.

If you want to see how the predictions did against the labels, you can run code like this:

```
score = 0
for item in range(0,99):
    prediction=np.argmax(predictions[item])
    label = test_labels[item]
    if prediction==label:
        score=score+1

print("Out of 100 predictions I got " + str(score) + " correct")
```

This should give you a score of 99 or 100 correct predictions.

You can also visualize the output of the model against the test data with this code:

```
for index in range(0,99):
    plt.figure(figsize=(6,3))
    plt.subplot(1,2,1)
    plot_image(index, predictions, test_labels, test_imgs)
    plt.show()
```

You can see some of the results of this in [Figure 12-3](#). (Note that all the code is available in the book's [GitHub repo](#), so if you need it, take a look there.)



Figure 12-3. Results of inference

This is just the plain, converted model without any optimizations for mobile added. In the next step you'll explore how to optimize this model for mobile devices.

Step 3. Optimize the Model

Now that you've seen the end-to-end process of training, converting, and using a model with the TensorFlow Lite interpreter, let's look at how to get started with optimizing and quantizing the model.

The first type of optimization, called *dynamic range quantization*, is achieved by setting the `optimizations` property on the converter, prior to performing the conversion. Here's the code:

```
converter = tf.lite.>TFLiteConverter.from_saved_model(CATS_VS_DOGS_SAVED_MODEL)
converter.optimizations = [tf.lite.>Optimize.DEFAULT]

tflite_model = converter.convert()
tflite_model_file = >'converted_model.tflite'

>with open(tflite_model_file, >'wb') >as f:
    f.write(tflite_model)
```

There are several optimization options available at the time of writing (more may be added later). These include:

OPTIMIZE_FOR_SIZE

Perform optimizations that make the model as small as possible.

OPTIMIZE_FOR_LATENCY

Perform optimizations that reduce inference time as much as possible.

DEFAULT

Find the best balance between size and latency.

In this case, the model size was close to 9 MB before this step but only 2.3 MB afterwards—a reduction of almost 70%. Various experiments have shown that models can be made up to 4× smaller, with a 2–3× speedup. Depending on the model type, however, there can be a loss in accuracy, so it's a good idea to test the model thoroughly if you quantize like this. In this case, I found that the accuracy of the model dropped from 99% to about 94%.

You can enhance this with *full integer quantization* or *float16 quantization* to take advantage of specific hardware. Full integer quantization changes the weights in the model from 32-bit floating point to 8-bit integer, which (particularly for larger models) can have a huge impact on model size and latency with a relatively small impact on accuracy.

To get full integer quantization, you'll need to specify a representative dataset that tells the convertor roughly what range of data to expect. Update the code as follows:

```
converter = tf.lite.TFLiteConverter.from_saved_model(CATS_VS_DOGS_SAVED_MODEL)

converter.optimizations = [tf.lite.Optimize.DEFAULT]

def representative_data_gen():
    for input_value, _ in test_batches.take(100):
        yield [input_value]

converter.representative_dataset = representative_data_gen
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]

tflite_model = converter.convert()
tflite_model_file = 'converted_model.tflite'

with open(tflite_model_file, "wb") as f:
    f.write(tflite_model)
```

Having this representative data allows the convertor to inspect data as it flows through the model and find where to best make the conversions. Then, by setting the supported ops (in this case to INT8), you can ensure that the precision is quantized in only those parts of the model. The result might be a slightly larger model—in this case, it went from 2.3 MB when using `converter.optimizations` only to 2.8 MB. However, the accuracy went back up to 99%. Thus, by following these steps you can reduce the model's size by about two-thirds, while maintaining its accuracy!

TensorFlow Lite for Microcontrollers

TensorFlow Lite for Microcontrollers is an ongoing and experimental build of TensorFlow Lite that is designed to run on devices with tiny amounts of memory (a field of machine learning often referred to as *TinyML*). It's a greatly reduced version of TensorFlow, given its expected runtime. It's written in C++ and runs on common microcontroller development boards such as the Arduino Nano, Sparkfun Edge, and more. To learn more about it, check out the book *TinyML: Machine Learning and TensorFlow Lite on Arduino and Ultra-Low Power Microcontrollers* by Pete Warden and Daniel Situnayake (O'Reilly).

Summary

In this chapter you got an introduction to TensorFlow Lite and saw how it is designed to get your models ready for running on smaller, lighter devices than your development environment. These include mobile operating systems like Android, iOS, and iPadOS, as well as mobile Linux-based computing environments like the Raspberry Pi and microcontroller-based systems that support TensorFlow. You built a simple model and used it to explore the conversion workflow. You then worked through a more complex example, using transfer learning to retrain an existing model for your dataset, converting it to TensorFlow Lite, and optimizing it for a mobile environment. In the next chapter you'll take this knowledge and explore how you can use the Android-based interpreter to use TensorFlow Lite in your Android apps.

Using TensorFlow Lite in Android Apps

Chapter 12 introduced you to TensorFlow Lite, a set of tools that help you convert your models into a format that can be consumed by mobile or embedded systems. Over the next few chapters you'll look into how to use those models on a variety of runtimes. Here, you'll see how to create Android apps that use TensorFlow Lite models. We'll start with a quick exploration of the main tool used to create Android apps: Android Studio.

What Is Android Studio?

Android Studio is an integrated development environment (IDE) for developing Android apps for a variety of devices, from phones and tablets to TVs, cars, watches, and more. In this chapter we'll focus on using it for phone apps. It's available to **download for free**, and there are versions for all major operating systems.

One of the nice things that Android Studio gives you is an Android emulator, so you can try out apps without needing to own a physical device. You'll be using that extensively in this chapter! Traditionally Android applications were built using the Java programming language, but recently Google introduced Kotlin to Android Studio, and you'll use that language in this chapter.

Kotlin?

Kotlin is a modern, open source language, which, along with Java, is the primary programming language for building Android apps. It's designed to be concise, reducing the amount of boilerplate code you need to write. It's also designed to be programmer-friendly, helping you avoid lots of common types of errors (such as null pointer exceptions, using built-in nullable types). It's also not an evolutionary dead end, being interoperable with preexisting libraries. This is particularly important

because Android has a history of Java language development, and thus many libraries have been built for Java.

Creating Your First TensorFlow Lite Android App

If you don't already have Android Studio, install it now. It can take a little while to get everything set up, updated, and ready to go. Over the next few pages, I'll step you through creating a new app, designing its user interface, adding TensorFlow Lite dependencies, and then coding it for inference. It will be a really simple app—one where you type in a value, and it performs inference and calculates $Y = 2X - 1$, where X is the value you entered. It's massive overkill for such simple functionality, but the scaffolding of an app like this is almost identical to that of a far more complex one.

Step 1. Create a New Android Project

Once you have Android Studio up and running, you can create a new app with File → New → New Project, which will open the Create New Project dialog (Figure 13-1).

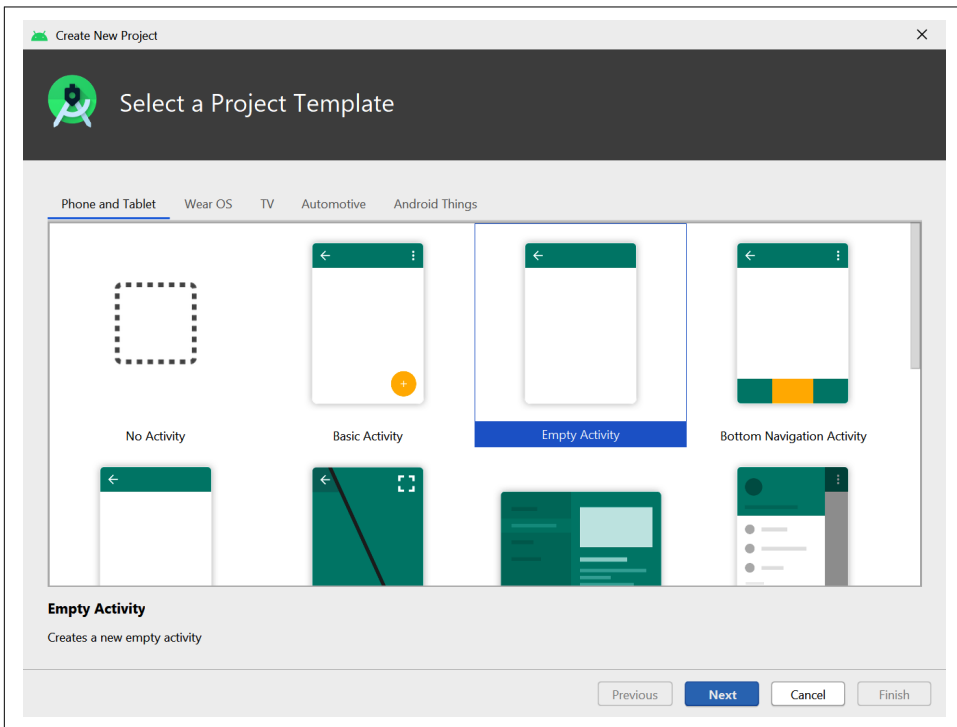


Figure 13-1. Creating a new project in Android Studio

Select the Empty Activity, as shown in [Figure 13-1](#). This is the simplest Android app, with very little preexisting code. Press Next and you'll be taken to the Configure Your Project dialog ([Figure 13-2](#)).

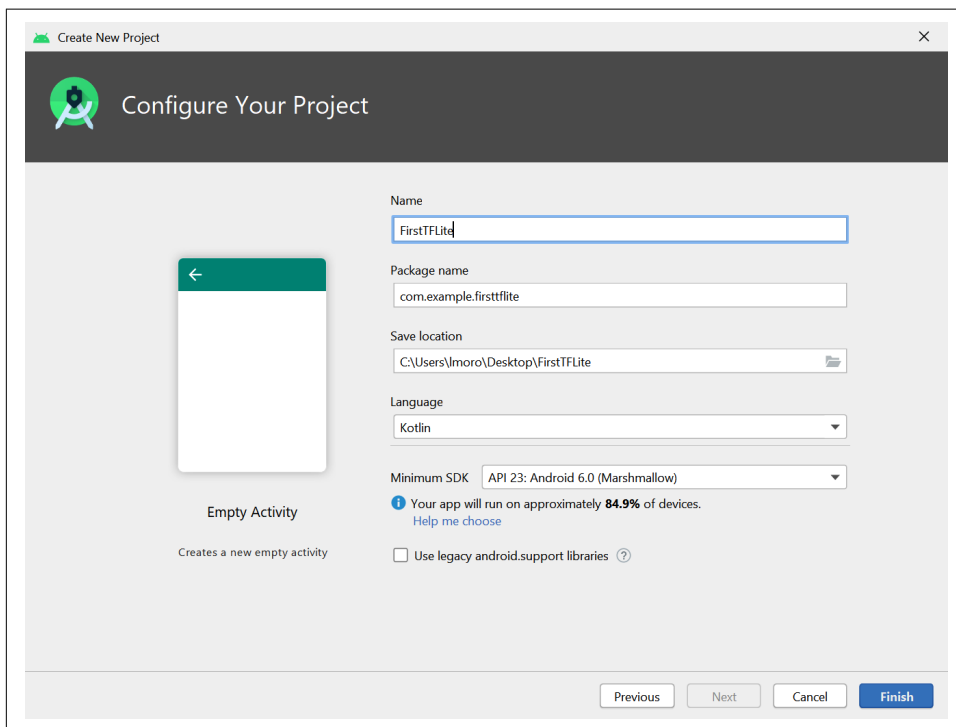


Figure 13-2. Configuring your project

In this dialog, set the name to be *FirstTFLite* as shown, and ensure that the language is Kotlin. The Minimum SDK level will probably default to API 23, and you can leave it at that if you like.

When you're done, press Finish. Android Studio will now create all the code for your app. Lots of files are needed for an Android application. The single activity you created has a layout file (in XML) that defines what it looks like, as well as a *.kt* (Kotlin) file for the associated source. There are also several configuration files defining how the app should be built, what dependencies it should use, and its resources, assets, and more. It can be quite overwhelming at first, even for a very simple app like this one.

Step 2. Edit Your Layout File

On the left side of your screen you'll see the project explorer. Make sure Android is selected at the top and find the *res* folder. Within it there's a *layout* folder, and within that you'll find *activity_main.xml* (see [Figure 13-3](#)).

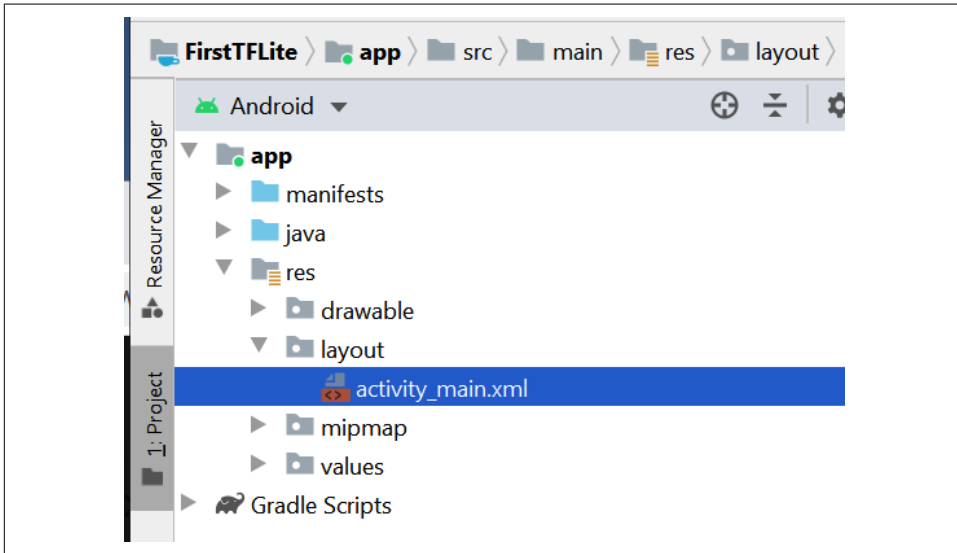


Figure 13-3. Finding your activity design file

Double-click this to open it, and you'll see the Android Studio Layout Editor. This gives you access to a visual representation of the user interface for your activity, as well as an XML editor that shows the definition. You may see just one or the other, but if you want to see both (which I recommend!) you can use the three buttons highlighted at the top right of [Figure 13-4](#). These give you (from left to right) the XML editor alone, a split screen with both the XML editor and the visual designer, and the visual designer by itself. Also note the Attributes tab directly underneath these. It allows you to edit the attributes of any of the individual user interface elements. As you build more Android apps you'll probably find it easier to use the visual layout tool to drag and drop items from the control palette onto the design surface and the Attributes window to set things like the layout width.

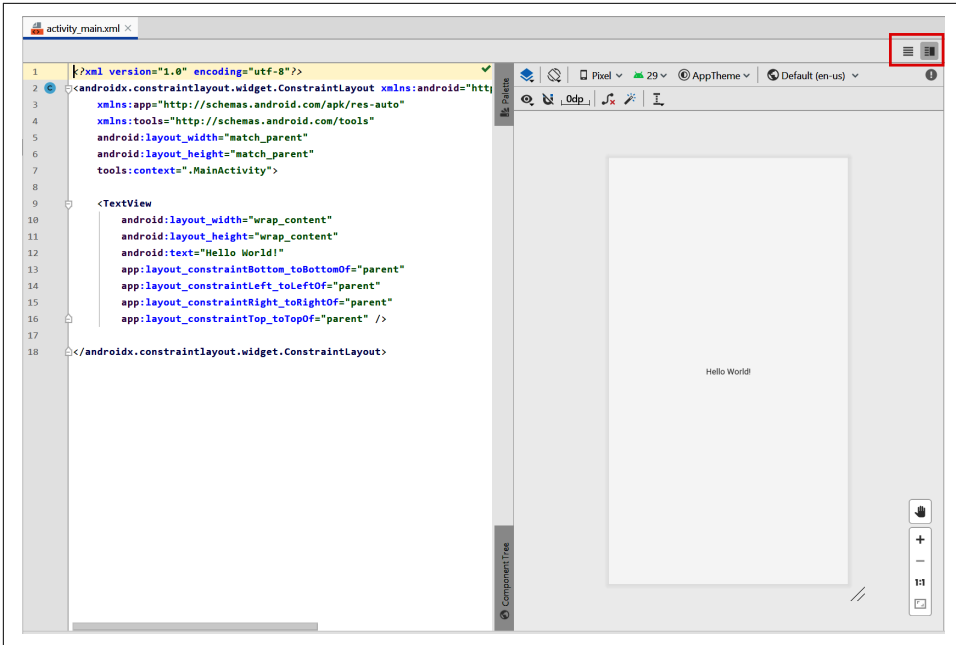


Figure 13-4. Using the Layout Editor in Android Studio

As you can see in [Figure 13-4](#), you'll have a very basic Android activity containing a single `TextView` control that says "Hello World." Replace all of the code for the activity with this:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:tools="http://schemas.android.com/tools"
    android:orientation="vertical"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_height="match_parent"
    android:layout_width="match_parent">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <TextView
            android:id="@+id/lblEnter"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Enter X: "
            android:textSize="18sp"></TextView>

        <EditText
            android:id="@+id/txtValue"
            android:layout_width="180dp"
```



```

        android:layout_height="wrap_content"
        android:inputType="number"
        android:text="1"></EditText>

        <Button
            android:id="@+id/convertButton"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Convert">

    </Button>
</LinearLayout>
</LinearLayout>

```

An important thing to note in this code is the `android:id` fields, particularly for the `EditText` and the `Button`. It's okay to change these, but if you do, you'll need to use the same values when you write your code a little later. I've called them `txtValue` and `convertButton` respectively, so watch out for those values in the code!

Step 3. Add the TensorFlow Lite Dependencies

TensorFlow Lite isn't natively part of the Android APIs, so when you use it in an Android app, you need to let the environment know that you'll be importing external libraries. In Android Studio this is achieved using the Gradle build tool. This tool lets you configure your environment by describing it with a JSON file called *build.gradle*. This can be a little confusing at first, particularly for new Android developers, because Android Studio actually gives you two Gradle files. Typically these are described as the "project-level" *build.gradle* and the "app-level" *build.gradle*. The first one is found within the project folder and the latter in the *app* folder (hence their names), as you can see in [Figure 13-5](#).

You are going to want to edit the app-level file, highlighted in [Figure 13-5](#). This has the dependency details for your app. Open it up, and make two edits. The first is to add an implementation to the dependencies section. This is to include the TensorFlow Lite libraries:

```
implementation 'org.tensorflow:tensorflow-lite:0.0.0-nightly'
```



You can get the latest version number for this dependency in the [TensorFlow Lite documentation](#).

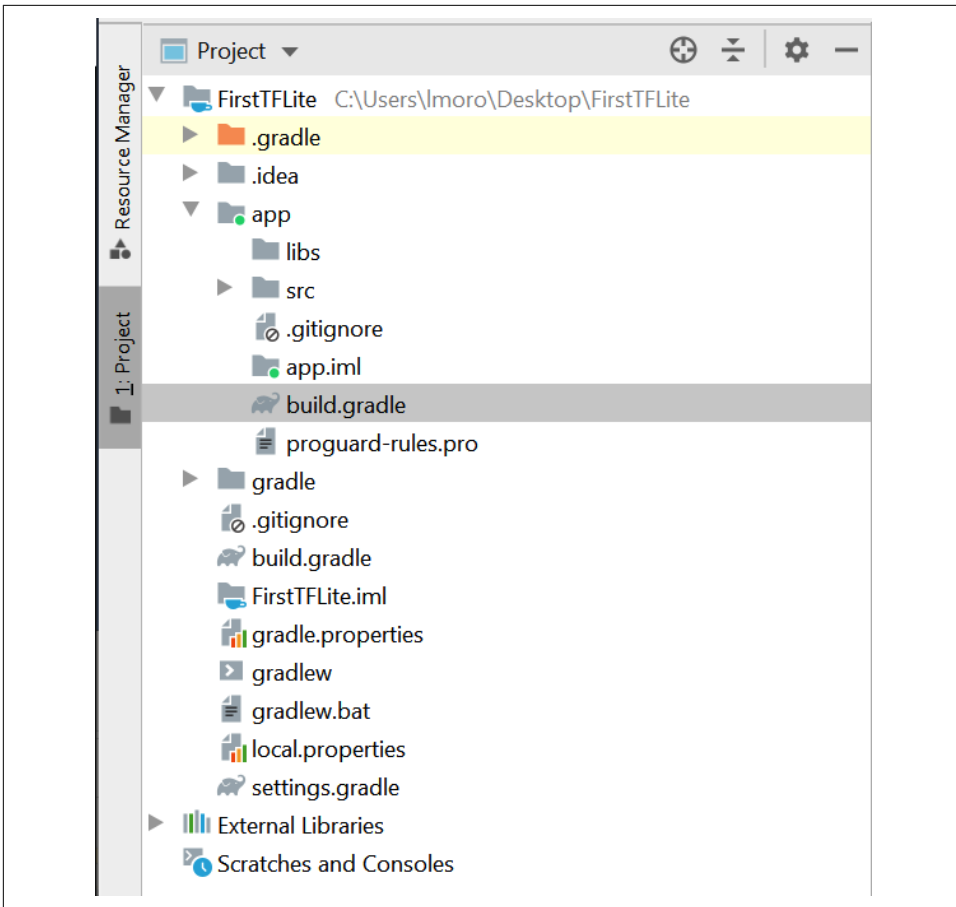


Figure 13-5. Selecting your *build.gradle* file

The second edit requires you to create a new setting within the `android{}` section, as follows:

```
android{
...
    aaptOptions {
        noCompress "tflite"
    }
...
}
```

This step prevents the compiler from compressing your *.tflite* file. The Android Studio compiler compiles assets to make them smaller so that the download time from the Google Play Store will be reduced. However, if the *.tflite* file is compressed, the TensorFlow Lite interpreter won't recognize it. To ensure that it doesn't get

compressed, you need to set `aaptOptions` to `noCompress` for `.tflite` files. If you used a different extension (some people just use `.lite`), make sure you have that here.

You can now try building your project. The TensorFlow Lite libraries will be downloaded and linked.

Step 4. Add Your TensorFlow Lite Model

In [Chapter 12](#) you created a very simple model that inferred $Y = 2X - 1$ from a set of X and Y values that it was trained on, converted it to TensorFlow Lite, and saved it as a `.tflite` file. You'll need that file for this step.

The first thing to do is create an `assets` folder in your project. To do this, navigate to the `app/src/main` folder in the project explorer, right-click on the `main` folder and select `New Directory`. Call it `assets`. Drag the `.tflite` file that you downloaded after training the model into that directory. If you didn't create this file earlier, you can find it in the book's [GitHub repository](#).

When you're done, the project explorer should look something like [Figure 13-6](#). Don't worry if the `assets` folder doesn't yet have the special assets icon; this will be updated by Android Studio eventually, typically after the next build.

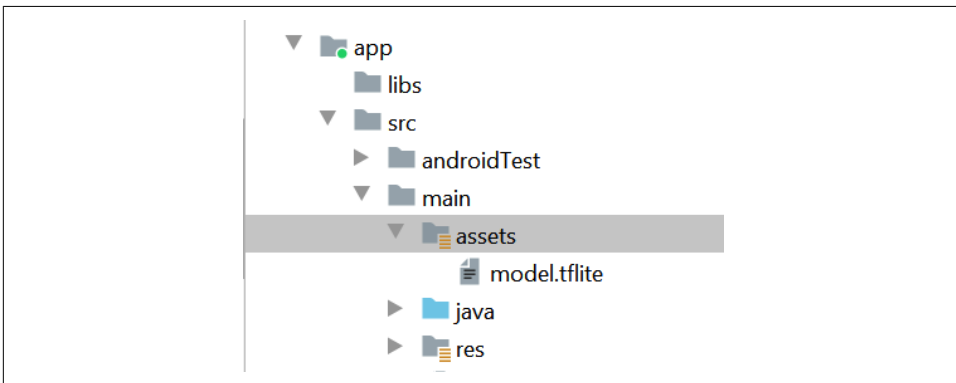


Figure 13-6. Adding your model as an asset

Now that all the plumbing is done, it's time to start coding!

Step 5. Write the Activity Code to Use TensorFlow Lite for Inference

Despite the fact that you're using Kotlin, your source files are in the `java` directory that you can see in [Figure 13-6](#). Open this, and you'll see a folder with your package name. Within that you should see your `MainActivity.kt` file. Double-click this file to open it in the code editor.

First, you'll need a helper function that loads the TensorFlow Lite model from the *assets* directory:

```
private fun loadModelFile(assetManager: AssetManager,
                          modelPath: String): ByteBuffer {
    val fileDescriptor = assetManager.openFd(modelPath)
    val inputStream = FileInputStream(fileDescriptor.fileDescriptor)
    val fileChannel = inputStream.channel
    val startOffset = fileDescriptor.startOffset
    val declaredLength = fileDescriptor.declaredLength
    return fileChannel.map(FileChannel.MapMode.READ_ONLY,
                          startOffset, declaredLength)
}
```

Because the *.tflite* file is effectively a compressed binary blob of weights and biases that the interpreter will use to build an internal neural network model, it's a *ByteBuffer* in Android terms. This code will load the file at *modelPath* and return it as a *ByteBuffer*.

Then, within your activity, at the class level (i.e., just below the class declaration, not within any class functions), you can add the declarations for the model and interpreter:

```
private lateinit var tflite : Interpreter
private lateinit var tflitemodel : ByteBuffer
```

So, in this case, the interpreter object that does all the work will be called *tflite* and the model that you'll load into the interpreter as a *ByteBuffer* is called *tflitemodel*.

Next, in the *onCreate* method, which gets called when the activity is created, add some code to instantiate the interpreter and load *model.tflite* into it:

```
try{
    tflitemodel = loadModelFile(this.assets, "model.tflite")
    tflite = Interpreter(tflitemodel)
} catch(ex: Exception){
    ex.printStackTrace()
}
```

Also, while you're in *onCreate*, add the code for the two controls that you'll interact with—the *EditText* where you'll type a value, and the *Button* that you'll press to get an inference:

```
var convertButton: Button = findViewById<Button>(R.id.convertButton)
convertButton.setOnClickListener{
    doInference()
}
txtValue = findViewById<EditText>(R.id.txtValue)
```

You'll also need to declare the `EditText` at the class level alongside `tflite` and `tflitemodel`, as it will be referred to within the next function. You can do that with the following:

```
private lateinit var txtValue : EditText
```

Finally, it's time to do the inference. You can do this with a new function called `doInference`:

```
private fun doInference(){  
}
```

Within this function you can gather the data from the input, pass it to TensorFlow Lite to get an inference, and then display the returned value.

The `EditText` control, where you'll enter the number, will provide you with a string, which you'll need to convert to a float:

```
var userVal: Float = txtValue.text.toString().toFloat()
```

As you'll recall from [Chapter 12](#), when feeding data into the model you need to format it as a Numpy array. Being a Python construct, Numpy isn't available in Android, but you can just use a `FloatArray` in this context. Even though you're only feeding in one value, it still needs to be in an array, roughly approximating a tensor:

```
var inputVal: FloatArray = floatArrayOf(userVal)
```

The model will return a stream of bytes to you that will need to be interpreted. As you know, you're getting a float value out of the model, and given that a float is 4 bytes, you can set up a `ByteBuffer` of 4 bytes to receive the output. There are several ways that bytes can be ordered, but you just need the default, native order:

```
var outputVal: ByteBuffer = ByteBuffer.allocateDirect(4)  
outputVal.order(ByteOrder.nativeOrder())
```

To perform the inference, you call the `run` method on the interpreter, passing it the input and output values. It will then read from the input value and write to the output value:

```
tflite.run(inputVal, outputVal)
```

The output is written to the `ByteBuffer`, whose pointer is now at the end of the buffer. To read it, you have to reset it to the beginning of the buffer:

```
outputVal.rewind()
```

And now you can read the contents of the `ByteBuffer` as a float:

```
var f:Float = outputVal.getFloat()
```

If you want to display this to the user, you can then use an `AlertDialog`:

```
val builder = AlertDialog.Builder(this)  
with(builder)
```

```

{
    setTitle("TFLite Interpreter")
    setMessage("Your Value is:$f")
    setNeutralButton("OK", DialogInterface.OnClickListener {
        dialog, id -> dialog.cancel()
    })
    show()
}

```

Now run the app and try it for yourself! You can see the results in [Figure 13-7](#).

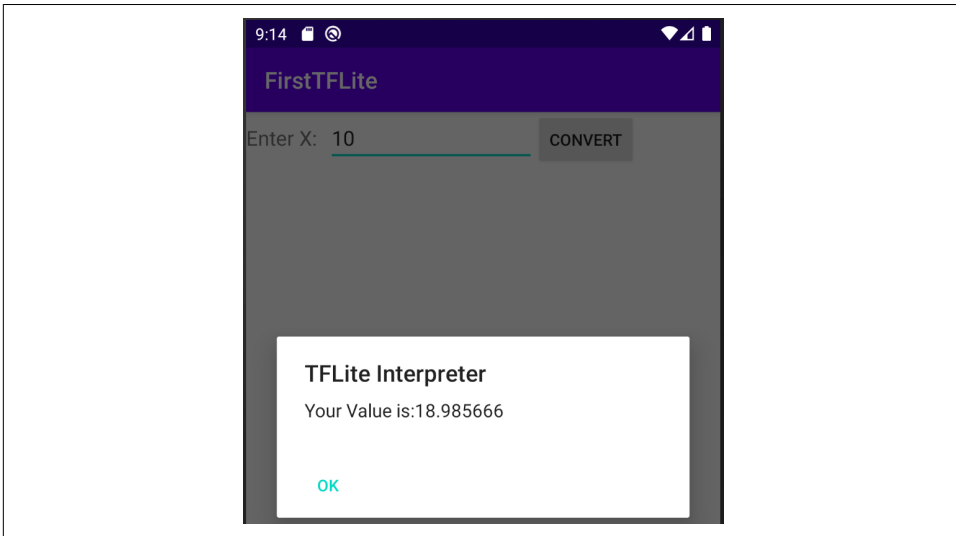


Figure 13-7. Running the interpreter in an emulator

Moving Beyond “Hello World”—Processing Images

As you saw over the last few pages, there’s a lot of scaffolding involved in building an Android app, and the TensorFlow Lite interpreter needs code and configuration in order to be properly initialized. Now that you have that out of the way, if you want to create other Android apps that use TensorFlow Lite, you’ll go through pretty much the same process. The only major difference you’ll encounter is in formatting your input data in a way that the model understands, and parsing output data in the same way. So, for example, in [Chapter 12](#) you built a Dogs vs. Cats model that allows you to feed in an image of a cat or a dog, and get an inference out. The model expects as input an image that is 224×224 pixels, in three color channels, and normalized—this requires figuring out how on earth to get an image from an Android image control and format it so that the neural network can understand it!

For example, let’s start with an image like that in [Figure 13-8](#), which is a simple image of a dog that happens to be 395×500 pixels.



Figure 13-8. Image of a dog to interpret

The first thing you need to do is resize it to 224×224 pixels, the image dimensions that the model was trained on. This can be done in Android using the `Bitmap` libraries. For example, you can create a new 224×224 bitmap with:

```
val scaledBitmap = Bitmap.createScaledBitmap(bitmap, 224, 224, false)
```

(In this case `bitmap` contains the raw image loaded as a resource by the app. The full app is available in the book's [GitHub repo](#).)

Now that it's the right size, you have to reconcile how the image is structured in Android with how the model expects it to be structured. If you recall, when training models earlier in the book you fed in images as normalized tensors of values. For example, an image like this would be $(224, 224, 3)$: 224×224 is the image size, and 3 is the color depth. The values were also all normalized to between 0 and 1.

So, in summary, you need $224 \times 224 \times 3$ float values between 0 and 1 to represent the image. To store that in a `ByteArray`, where 4 bytes make a float, you can use this code:

```
val byteBuffer = ByteBuffer.allocateDirect(4 * 224 * 224 * 3)
byteBuffer.order(ByteOrder.nativeOrder())
```

Our Android image, on the other hand, has each pixel stored as a 32-bit integer in an RGB value. This might look something like `0x0010FF10` for a particular pixel. The first two values are the transparency, which you can ignore, and the rest are for RGB; i.e., `0x10` for red, `0xFF` for green, and `0x10` for blue. The simple normalization you've been doing to this point is just to divide the R, G, B channel values by 255, which would give you `.06275` for red, `1` for green, and `.06275` for blue.

So, to do this conversion, let's first turn our bitmap into an array of 224×224 integers, and copy the pixels in. You can do this using the `getPixels` API:

```
val intValues = IntArray(224 * 224)
scaledbitmap.getPixels(intValues, 0, 224, 0, 0, 224, 224)
```

Now you'll need to iterate through this array, reading the pixels one by one and converting them into normalized floats. You'll use bit shifting to get the particular channels. For example, consider the value `0x0010FF10` from earlier. If you shift that by 16 bits to the right, you'll get `0x0010` (with the `FF10` being "lost"). If you then "and" that by `0xFF`, you'll get `0x10`, keeping just the bottom two numbers. Similarly, if you had shifted by 8 bits to the right you'd have `0x0010FF`, and performing an "and" on that would give you `0xFF`. It's a technique that allows you to quickly and easily strip out the relevant bits that make up the pixels. You can use the `shr` operation on an integer for this, with `input.shr(16)` reading "shift input 16 pixels to the right":

```
var pixel = 0
for (i in 0 until INPUT_SIZE) {
    for (j in 0 until INPUT_SIZE) {
        val input = intValues[pixel++]
        byteBuffer.putFloat(((input.shr(16) and 0xFF) / 255))
        byteBuffer.putFloat(((input.shr(8) and 0xFF) / 255))
        byteBuffer.putFloat(((input and 0xFF) / 255))
    }
}
```

As before, when it comes to the output, you need to define an array to hold the result. It doesn't *have* to be a `ByteArray`; indeed, you can define something like a `FloatArray` if you know the results are going to be floats, as they usually are. In this case, with the Dogs vs. Cats model, you have two labels, and the model architecture was defined with two neurons in the output layer, containing the respective properties for the classes cat and dog. So, to read back the results you can define a structure to contain the output tensor like this:

```
val result = Array(1) { FloatArray(2) }
```

Note that it's a single array that contains an array of two items. Remember back when using Python you might see a value like `[[1.0 0.0]]`—it's the same here. The `Array(1)` is defining the containing array `[]`, while the `FloatArray(2)` is the `[1.0 0.0]`. It can be a little confusing, for sure, but it's something that I hope you'll get used to as you write more TensorFlow apps!

As before, you interpret using `interpreter.run`:

```
interpreter.run(byteBuffer, result)
```

And now your result will be an array, containing an array of two values. You can see what it looks like in the Android debugger in [Figure 13-8](#).

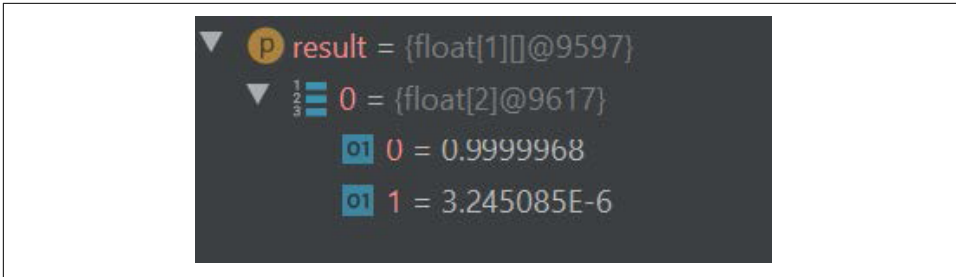


Figure 13-9. Parsing the output value

As you create mobile apps with Android, this is the most complex part—other than creating the model, of course—that you’ll have to take into account. How Python represents values, particularly with Numpy, can be very different from how Android does. You’ll have to create convertors to reformat your data for how neural networks expect the data to be input, and you’ll have to understand the output schema that the neural network uses so that you can parse the results.

Code Generation

At the time of writing, a [tool for generating code from metadata](#) is available in experimental mode. To use it, you need to add metadata to your TensorFlow Lite model when you perform the conversion. I won’t go into full details here as it’s evolving, but you can consult the [documentation](#) to see how to create the metadata for your model and then use this tool to generate code, helping you avoid dealing with low-level Byte Buffers like the ones you’ve been using in this chapter. You may also want to take a look at the [TensorFlow Lite Model Maker library](#).

TensorFlow Lite Sample Apps

The TensorFlow team provides many open source sample apps that you can dissect to learn how they work from the foundations you’ve built up in this chapter. They include (but are not limited to) the following:

Image classification

Read input from the device’s camera and classify up to a thousand different items.

Object detection

Read input from the device’s camera and give bounding boxes to objects that are detected.

Pose estimation

Take a look at the figures in the camera and infer their poses.

Speech recognition

Recognize common verbal commands.

Gesture recognition

Train a model for hand gestures and recognize them in the camera.

Smart reply

Take input messages and generate replies to them.

Image segmentation

Similar to object detection, but predict which class each pixel in an image belongs to.

Style transfer

Apply new art styles to any image.

Digit classifier

Recognize handwritten digits.

Text classification

Using a model trained on the IMDb dataset, recognize sentiment in text.

Question answering

Using Bidirectional Encoder Representations from Transformers (BERT), answer user queries automatically!

You can find another curated list of apps on GitHub in the [Awesome TFLite repo](#).

The TensorFlow Lite Android Support Library

The TensorFlow team has also created a [support library for TensorFlow Lite](#) whose goal is to provide a set of high-level classes to support common scenarios on Android. At the time of writing, it provides the ability to handle some computer vision scenarios by supplying premade classes and functions to deal with the complexities of using images as tensors, as well as parsing probability arrays for output.

Summary

In this chapter you got a taste of using TensorFlow Lite on Android. You were introduced to the anatomy of an Android application and how you can weave TensorFlow Lite into it. You learned how to implement a model as an Android asset, and how to load and use that in an interpreter. Most importantly, you saw the need for converting your Android-based data (such as images or numbers) into input arrays that emulate the tensors used in the model, and how to parse output data, realizing that it too is effectively memory-mapped tensors in ByteBuffers. You stepped in detail through a couple of examples that showed how to do this, which hopefully has equipped you to be able to handle other scenarios. In the next chapter you'll do this all over again, but this time on iOS with Swift.

Using TensorFlow Lite in iOS Apps

Chapter 12 introduced you to TensorFlow Lite and how you can use it to convert your TensorFlow models into a power-efficient, compact format that can be used on mobile devices. In **Chapter 13** you then explored creating Android apps that use TensorFlow Lite models. In this chapter you'll do the same thing but with iOS, creating a couple of simple apps, and seeing how you can do inference on a TensorFlow Lite model using the Swift programming language.

You'll need a Mac if you want to follow along with the examples in this chapter, as the development tool to use is Xcode, which is only available on Mac. If you don't have it already, you can install it from the App Store. It will give you everything you need, including an iOS Simulator on which you can run iPhone and iPod apps without a physical device.

Creating Your First TensorFlow Lite App with Xcode

Once you have Xcode up and running, you can follow the steps outlined in this section to create a simple iOS app that incorporates the $Y = 2X - 1$ model from **Chapter 12**. While it's an extremely simple scenario, and definite overkill for a machine learning app, the skeleton structure is the same as that used for more complex apps, and I've found it a useful way of demonstrating how to use models in an app.

Step 1. Create a Basic iOS App

Open Xcode and select File → New Project. You'll be asked to pick the template for your new project. Choose Single View App, which is the simplest template (**Figure 14-1**), and click Next.

After that you'll be asked to choose options for your new project, including a name for the app. Call it *firstlite*, and make sure that the language is Swift and the user interface is Storyboard (Figure 14-2).

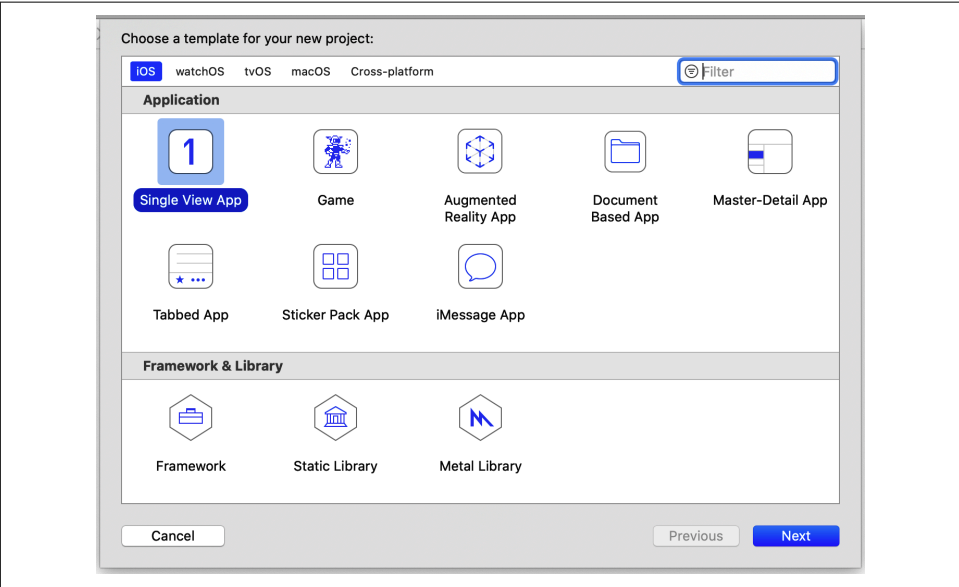


Figure 14-1. Creating a new iOS application in Xcode

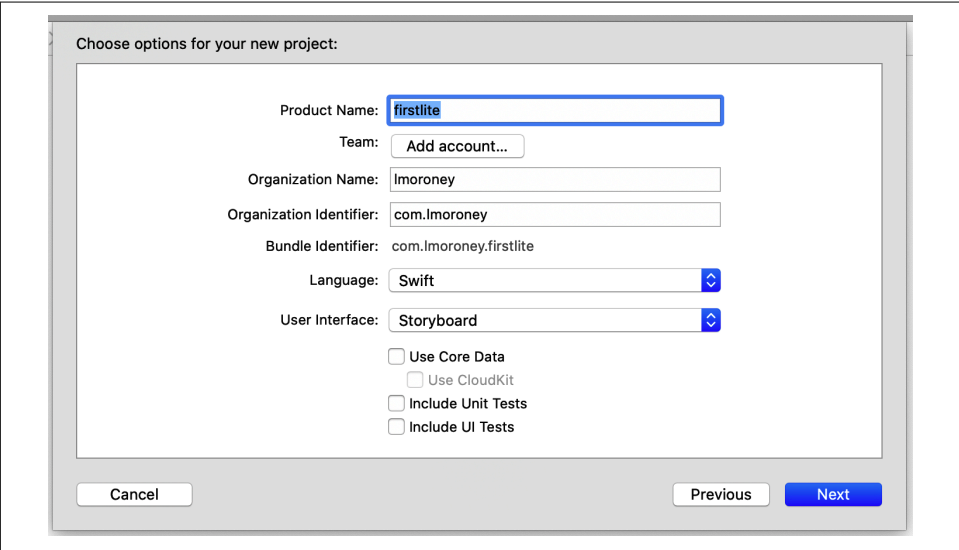


Figure 14-2. Choosing options for your new project

Click Next to create a basic iOS app that will run on an iPhone or iPad simulator. The next step is to add TensorFlow Lite to it.

Step 2. Add TensorFlow Lite to Your Project

To add dependencies to an iOS project, you can use a technology called **CocoaPods**, a dependency management project with many thousands of libraries that can be easily integrated into your app. To do so, you create a specification called a Podfile, which contains details about your project and the dependencies you want to use. This is a simple text file called *Podfile* (no extension), and you should put it in the same directory as the *firstlite.xcodeproj* file that was created for you by Xcode. Its contents should be as follows:

```
# Uncomment the next line to define a global platform for your project
platform :ios, '12.0'

target 'firstlite' do
  # Comment the next line if you're not using Swift and don't want to
  # use dynamic frameworks
  use_frameworks!

  # Pods for ImageClassification
  pod 'TensorFlowLiteSwift'
end
```

The important part is the line that reads `pod 'TensorFlowLiteSwift'`, which indicates that the TensorFlow Lite Swift libraries need to be added to the project.

Next, using Terminal, change to the directory containing the Podfile and issue the following command:

```
pod install
```

The dependencies will be downloaded and added to your project, stored in a new folder called *Pods*. You'll also have an *.xcworkspace* file added, as shown in **Figure 14-3**. Use this one in the future to open your project, and not the *.xcodeproj* file.

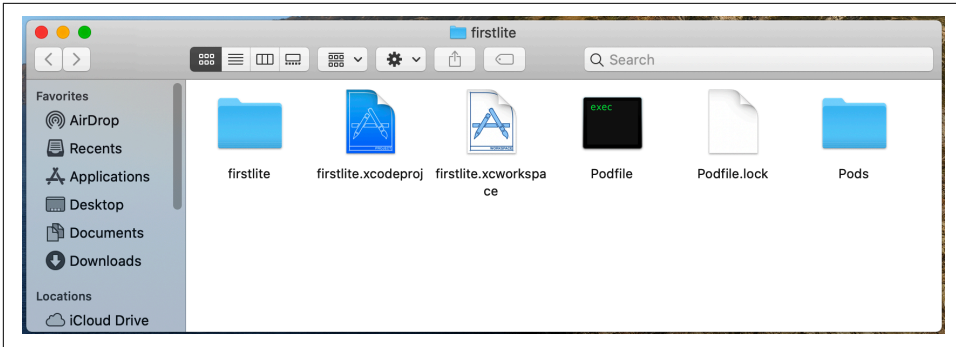


Figure 14-3. Your file structure after running `pod install`

You now have a basic iOS app, and you have added the TensorFlow Lite dependencies. The next step is to create your user interface.

Step 3. Create the User Interface

The Xcode storyboard editor is a visual tool that allows you to create a user interface. After opening your workspace, you'll see a list of source files on the left. Select *Main.storyboard*, and using the controls palette, you can drag and drop controls onto the view for an iPhone screen (Figure 14-4).

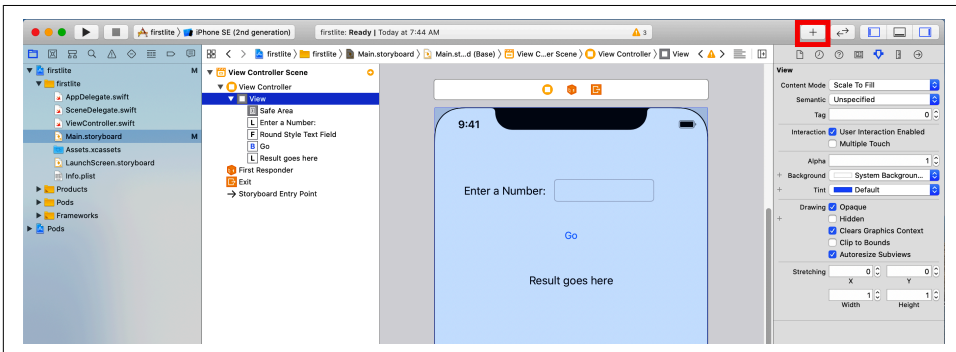


Figure 14-4. Adding controls to the storyboard

If you can't find the controls palette, you can access it by clicking the + at the top right of the screen (highlighted in Figure 14-4). Using it, add a Label, and change the text to “Enter a Number.” Then add another one with the text “Result goes here.” Add a Button and change its caption to “Go,” and finally add a Text Field. Arrange them similarly to what you can see in Figure 14-4. It doesn't have to be pretty!

Now that the controls are laid out, you want to be able to refer to them in code. In storyboard parlance you do this using either *outlets* (when you want to address the

control to read or set its contents) or *actions* (when you want to execute some code when the user interacts with the control).

The easiest way to wire this up is to have a split screen, with the storyboard on one side and the *ViewController.swift* code that underlies it on the other. You can achieve this by selecting the split screen control (highlighted in Figure 14-5), clicking on one side and selecting the storyboard, and then clicking on the other side and selecting *ViewController.swift*.

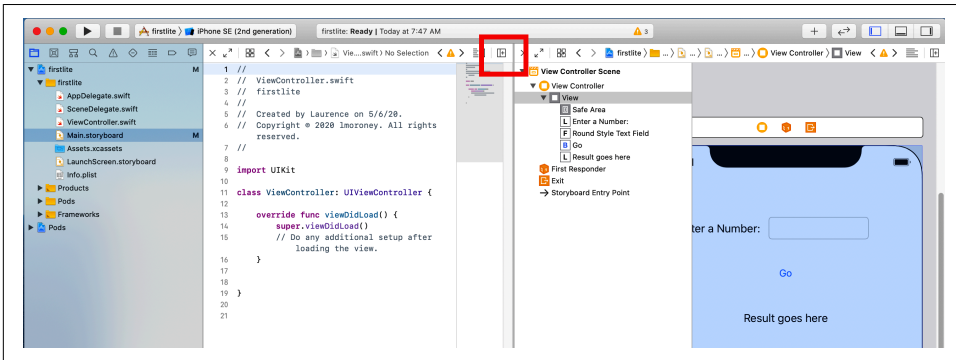


Figure 14-5. Splitting the screen

Once you’ve done this, you can start creating your outlets and actions by dragging and dropping. With this app, the user types a number into the text field, presses Go, and then runs inference on the value they typed. The result will be rendered in the label that says “Result goes here.”

This means you’ll need to read or write to two controls, reading the contents of the text field to get what the user typed in, and writing the result to the “Results goes here” label. Thus, you’ll need two outlets. To create them, hold down the Ctrl key and drag the control on the storyboard onto the *ViewController.swift* file, dropping it just below the class definition. A pop-up will appear asking you to define it (Figure 14-6).

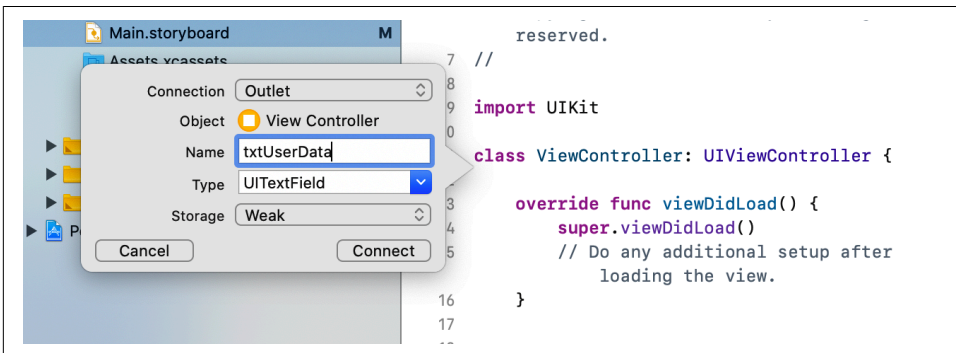


Figure 14-6. Creating an outlet

Ensure the connection type is Outlet, and create an outlet for the text field called `txtUserData` and one for the label called `txtResult`.

Next, drag the button over to the `ViewController.swift` file. In the pop-up, ensure that the connection type is Action and the event type is Touch Up Inside. Use this to define an action called `btnGo` (Figure 14-7).

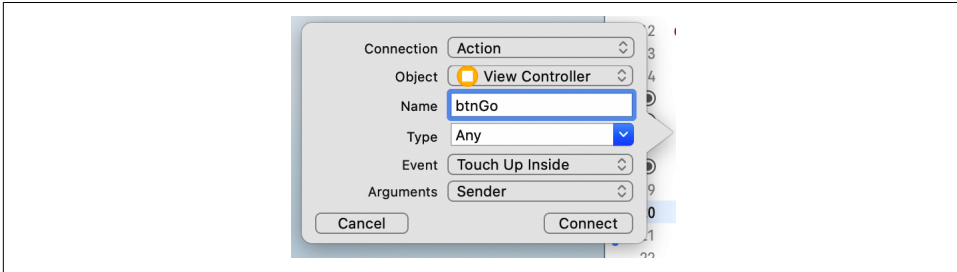


Figure 14-7. Adding an action

At this point your `ViewController.swift` file should look like this—note the `IBOutlet` and `IBAction` code:

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var txtUserData: UITextField!

    @IBOutlet weak var txtResult: UILabel!
    @IBAction func btnGo(_ sender: Any) {
    }
    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.
    }
}
```

Now that the UI is squared away, the next step will be to create the code that will handle the inference. Instead of having this in the same Swift file as the `ViewController` logic, you'll place it in a separate code file.

Step 4. Add and Initialize the Model Inference Class

To keep the UI separate from the underlying model inference, you'll create a new Swift file containing a `ModelParser` class. This is where all the work of getting the data into the model, running the inference, and then parsing the results will happen. In Xcode, choose `File` → `New File` and select `Swift File` as the template type (Figure 14-8).

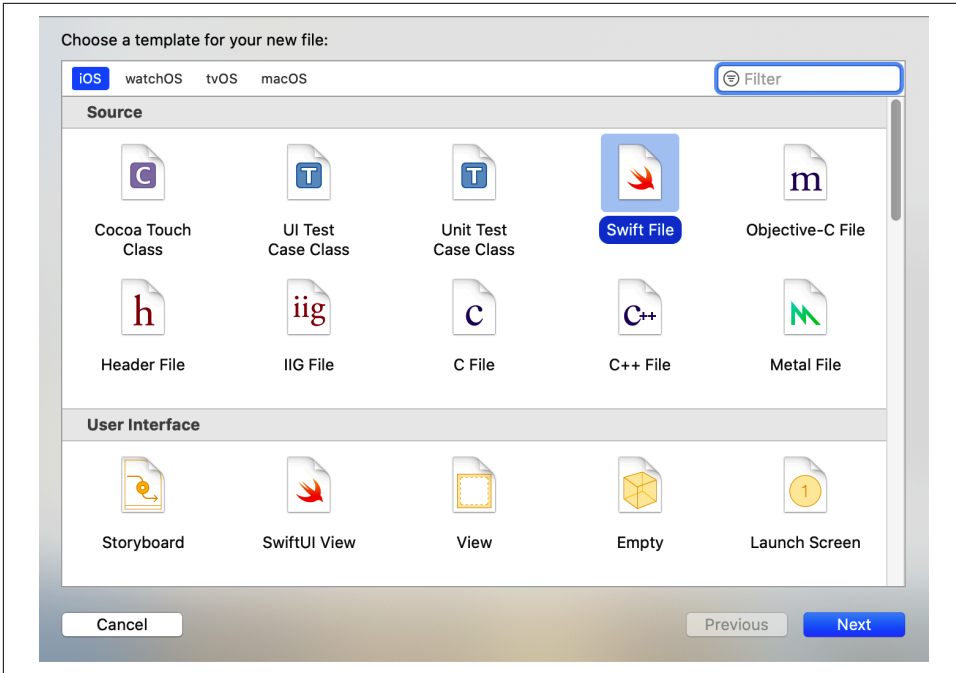


Figure 14-8. Adding a new Swift file

Call this *ModelParser*, and ensure that the checkbox targeting it to the firstlite project is checked (Figure 14-9).

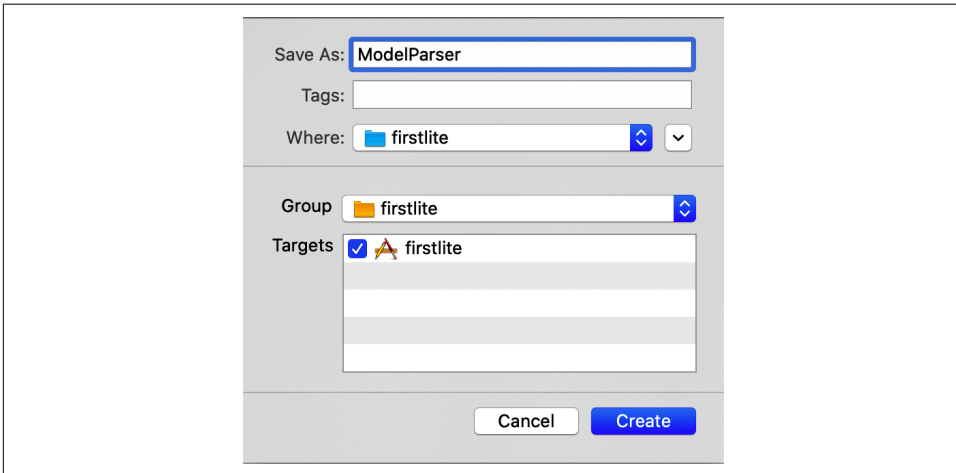


Figure 14-9. Adding *ModelParser.swift* to your project

This will add a *ModelParser.swift* file to your project that you can edit to add the inference logic. First, ensure that the imports at the top of the file include TensorFlowLite:

```
import Foundation
import TensorFlowLite
```

You'll pass a reference to the model file, *model.tflite*, to this class—you haven't added it yet, but you will soon:

```
typealias FileInfo = (name: String, extension: String)

enum ModelFile {
    static let modelInfo: FileInfo = (name: "model", extension: "tflite")
}
```

This `typealias` and `enum` make the code a little more compact. You'll see them in use in a moment. Next you'll need to load the model into an interpreter, so first declare the interpreter as a private variable to the class:

```
private var interpreter: Interpreter
```

Swift requires variables to be initialized, which you can do within an `init` function. The following function will take two input parameters. The first, `modelFileInfo`, is the `FileInfo` type you just declared. The second, `threadCount`, is the number of threads to use to initialize the interpreter, which we'll set to 1. Within this function you'll create a reference to the model file that you described earlier (*model.tflite*):

```
init?(modelFileInfo: FileInfo, threadCount: Int = 1) {
    let modelFilename = modelFileInfo.name

    guard let modelPath = Bundle.main.path
    (
        forResource: modelFilename,
        ofType: modelFileInfo.extension
    )
    else {
        print("Failed to load the model file")
        return nil
    }
}
```

Once you have the path to the model file in the bundle, you can load it:

```
do
{
    interpreter = try Interpreter(modelPath: modelPath)
}
catch let error
{
    print("Failed to create the interpreter")
    return nil
}
```

Step 5. Perform the Inference

Within the `ModelParser` class, you can then do the inference. The user will type a string value in the text field, which will be converted to a float, so you'll need a function that takes a float, passes it to the model, runs the inference, and parses the return value.

Start by creating a function called `runModel`. Your code will need to catch errors, so start it with a `do{`:

```
func runModel(withInput input: Float) -> Float? {  
    do{
```

Next, you'll need to allocate tensors on the interpreter. This initializes it and readies it for inference:

```
        try interpreter.allocateTensors()
```

Then you'll create the input tensor. As Swift doesn't have a `Tensor` data type, you'll need to write the data directly to memory in an `UnsafeMutableBufferPointer`. You can specify the type of this, which will be `Float`, and write one value (as you only have one float), starting from the address of the variable called `data`. This will effectively copy all the bytes for the float into the buffer:

```
        var data: Float = input  
        let buffer: UnsafeMutableBufferPointer<Float> =  
            UnsafeMutableBufferPointer(start: &data, count: 1)
```

With the data in the buffer, you can then copy it to the interpreter at input 0. You only have one input tensor, so you can specify it as the buffer:

```
        try interpreter.copy(Data(buffer: buffer), toInputAt: 0)
```

To execute the inference, you invoke the interpreter:

```
        try interpreter.invoke()
```

There's only one output tensor, so you can read it by taking the output at 0:

```
        let outputTensor = try interpreter.output(at: 0)
```

Similar to when inputting the values, you're dealing with low-level memory, which is unsafe data. It's in an array of `Float32` values (it only has one element but still needs to be treated as an array), which can be read like this:

```
        let results: [Float32] =  
            [Float32](unsafeData: outputTensor.data) ?? []
```

If you're not familiar with the `??` syntax, this says to make the results an array of `Float32` by copying the output tensor into it, and if that fails, to make it an empty array. For this code to work, you'll need to implement an `Array` extension; the full code for that will be shown in a moment.

Once you have the results in an array, the first element will be your result. If this fails, just return `nil`:

```
guard let result = results.first else {
    return nil
}
return result
}
```

The function began with a `do{`, so you'll need to catch any errors, print them, and return `nil` in that event:

```
catch {
    print(error)
    return nil
}
}
```

Finally, still in *ModelParser.swift*, you can add the `Array` extension that handles the unsafe data and loads it into an array:

```
extension Array {
    init?(unsafeData: Data) {
        guard unsafeData.count % MemoryLayout<Element>.stride == 0
            else { return nil }
        #if swift(>=5.0)
        self = unsafeData.withUnsafeBytes {
            .init($0.bindMemory(to: Element.self))
        }
        #else
        self = unsafeData.withUnsafeBytes {
            .init(UnsafeBufferPointer<Element>(
                start: $0,
                count: unsafeData.count / MemoryLayout<Element>.stride
            ))
        }
        #endif // swift(>=5.0)
    }
}
```

This is a handy helper that you can use if you want to parse floats directly out of a TensorFlow Lite model.

Now that the class for parsing the model is done, the next step is to add the model to your app.

Step 6. Add the Model to Your App

To add the model to your app, you'll need a *models* directory within the app. In Xcode, right-click on the *firstlite* folder and select New Group (Figure 14-10). Call the new group *models*.

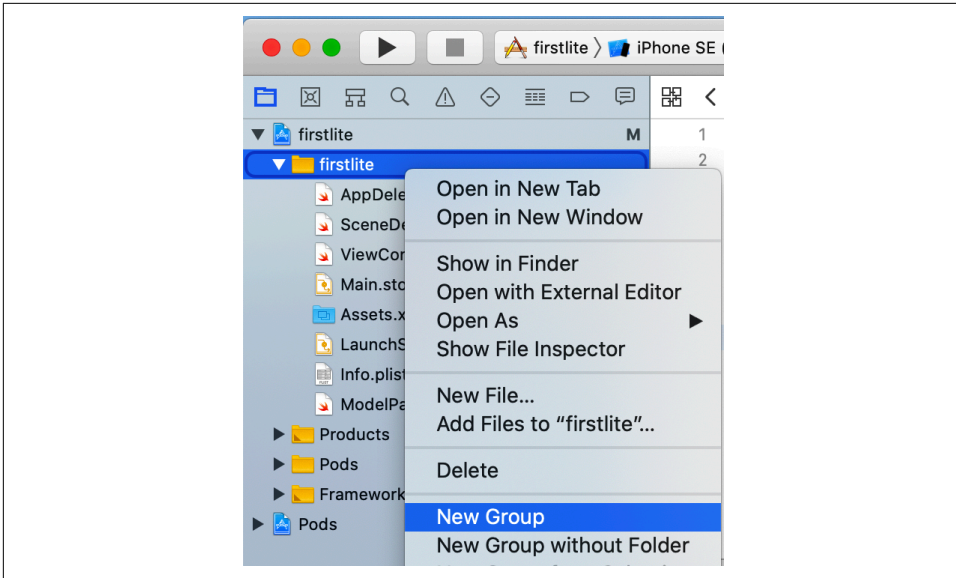


Figure 14-10. Adding a new group to your app

You can get the model by training the simple $Y = 2X - 1$ sample from [Chapter 12](#). If you don't have it already, you can use the Colab in the book's [GitHub repository](#).

Once you have the converted model file (called *model.tflite*), you can drag and drop it into Xcode on the models group you just added. Select “Copy items if needed” and ensure you add it to the target firstlite by checking the box beside it ([Figure 14-11](#)).

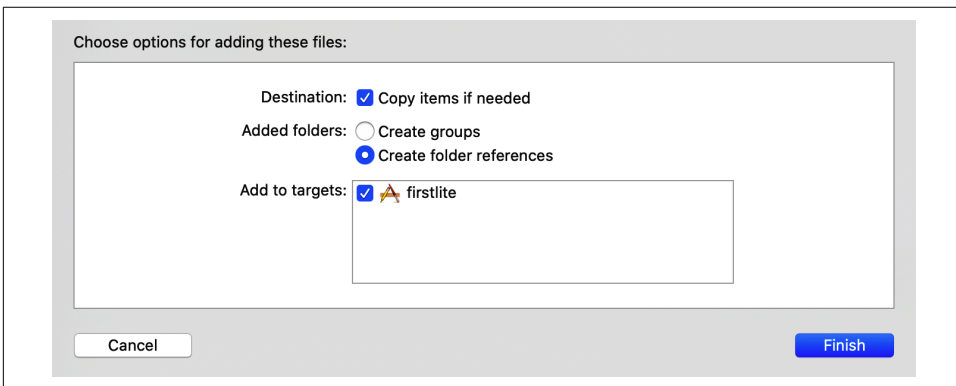


Figure 14-11. Adding the model to your project

The model will now be in your project and available for inference. The final step is to complete the user interface logic—then you'll be ready to go!

Step 7. Add the UI Logic

Earlier, you created the storyboard containing the UI description and began editing the *ViewController.swift* file containing the UI logic. As most of the work of inference has now been offloaded to the *ModelParser* class, the UI logic should be very light.

Start by adding a private variable declaring an instance of the *ModelParser* class:

```
private var modelParser: ModelParser? =  
    ModelParser(modelFileInfo: ModelFile.modelInfo)
```

Previously, you created an action on the button called *btnGo*. This will be called when the user touches the button. Update that to execute a function called *doInference* when the user takes that action:

```
@IBAction func btnGo(_ sender: Any) {  
    doInference()  
}
```

Next you'll construct the *doInference* function:

```
private func doInference() {
```

The text field that the user will enter data into is called *txtUserData*. Read this value, and if it's empty just set the result to *0.00* and don't bother with any inference:

```
    guard let text = txtUserData.text, text.count > 0 else {  
        txtResult.text = "0.00"  
        return  
    }
```

Otherwise, convert it to a float. If this fails, exit the function:

```
    guard let value = Float(text) else {  
        return  
    }
```

If the code has reached this point, you can now run the model, passing it that input. The *ModelParser* will do the rest, returning you either a result or *nil*. If the return value is *nil*, then you'll exit the function:

```
    guard let result = self.modelParser?.runModel(withInput: value) else {  
        return  
    }
```

Finally, if you've reached this point, you have a result, so you can load it into the label (called *txtResult*) by formatting the float as a string:

```
    txtResult.text = String(format: "%.2f", result)
```

That's it! The complexity of the model loading and inference has been handled by the *ModelParser* class, keeping your *ViewController* very light. For convenience, here's the complete listing:

```

import UIKit

class ViewController: UIViewController {
    private var modelParser: ModelParser? =
        ModelParser(modelFileInfo: ModelFile.modelInfo)
    @IBOutlet weak var txtUserData: UITextField!

    @IBOutlet weak var txtResult: UILabel!
    @IBAction func btnGo(_ sender: Any) {
        doInference()
    }
    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.
    }
    private func doInference() {

        guard let text = txtUserData.text, text.count > 0 else {
            txtResult.text = "0.00"
            return
        }
        guard let value = Float(text) else {
            return
        }
        guard let result = self.modelParser?.runModel(withInput: value) else {
            return
        }
        txtResult.text = String(format: "%.2f", result)
    }
}

```

You’ve now done everything you need to get the app working. Run it, and you should see it in the simulator. Type a number in the text field, press the button, and you should see a result in the results field, as shown in [Figure 14-12](#).

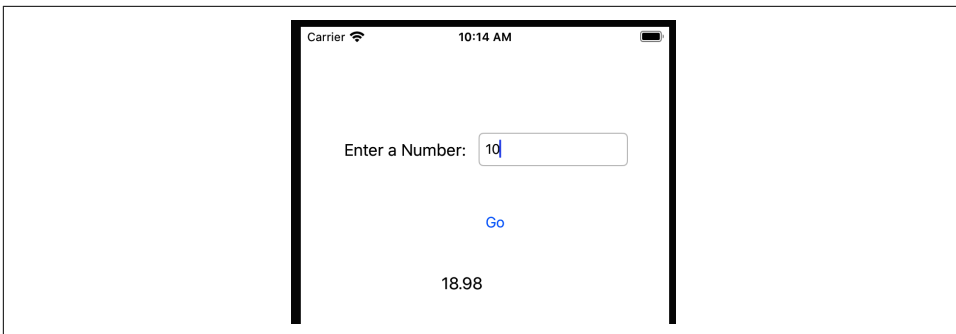


Figure 14-12. Running the app in the iPhone Simulator

While this was a long journey for a very simple app, it should provide a good template to help you understand how TensorFlow Lite works. In this walkthrough you saw how to:

- Use pods to add the TensorFlow Lite dependencies.
- Add a TensorFlow Lite model to your app.
- Load the model into an interpreter.
- Access the input tensors, and write directly to their memory.
- Read the memory from the output tensors and copy that to high-level data structures like float arrays.
- Wire it all up to a user interface with a storyboard and view controller.

In the next section, you'll move beyond this simple scenario and look at handling more complex data.

Moving Beyond “Hello World”—Processing Images

In the previous example you saw how to create a full app that uses TensorFlow Lite to do very simple inference. However, despite the simplicity of the app, the process of getting data into the model and parsing data out of the model can be a little unintuitive because you're handling low-level bits and bytes. As you get into more complex scenarios, such as managing images, the good news is that the process isn't that much more complicated.

Consider the Dogs vs. Cats model you created in [Chapter 12](#). In this section you'll see how to create an iOS app in Swift with a trained model that, given an image of a cat or a dog, will be able to infer what is in the picture. The full app code is available in the [GitHub repo](#) for this book.

First, recall that the tensor for an image has three dimensions: width, height, and color depth. So, for example, when using the MobileNet architecture that the Dogs vs. Cats mobile sample is based on, the dimensions are $224 \times 224 \times 3$ —each image is 224×224 pixels and has 3 bytes for color depth. Note that each pixel is represented by a value between 0 and 1 indicating the intensity of that pixel on the red, green, and blue channels.

In iOS, images are typically represented as instances of the `UIImage` class, which has a useful `pixelBuffer` property that returns a buffer of all the pixels in the image.

Within the `CoreImage` libraries, there's a `CVPixelBufferGetPixelFormatType` API that will return the type of the pixel buffer:

```
let sourcePixelFormat = CVPixelBufferGetPixelFormatType(pixelBuffer)
```

This will typically be a 32-bit image with channels for alpha (aka transparency), red, green, and blue. However, there are multiple variants, generally with these channels in different orders. You'll want to ensure that it's one of these formats, as the rest of the code won't work if the image is stored in a different format:

```
assert(sourcePixelFormat == kCVPixelFormatType_32ARGB ||
       sourcePixelFormat == kCVPixelFormatType_32BGRA ||
       sourcePixelFormat == kCVPixelFormatType_32RGBA)
```

As the desired format is 224×224 , which is square, the best thing to do next is to crop the image to the largest square in its center, using the `centerThumbnail` property, and then scale this down to 224×224 :

```
let scaledSize = CGSize(width: 224, height: 224)
guard let thumbnailPixelBuffer =
    pixelBuffer.centerThumbnail(ofSize: scaledSize)
else {
    return nil
}
```

Now that you have the image resized to 224×224 , the next step is to remove the alpha channel. Remember that the model was trained on $224 \times 224 \times 3$, where the 3 is the RGB channels, so there is no alpha.

Now that you have a pixel buffer, you need to extract the RGB data from it. This helper function achieves that for you by finding the alpha channel and slicing it out:

```
private func rgbDataFromBuffer(_ buffer: CVPixelBuffer,
                              byteCount: Int) -> Data? {

    CVPixelBufferLockBaseAddress(buffer, .readOnly)
    defer { CVPixelBufferUnlockBaseAddress(buffer, .readOnly) }
    guard let mutableRawPointer =
        CVPixelBufferGetBaseAddress(buffer)
    else {
        return nil
    }

    let count = CVPixelBufferGetDataSize(buffer)
    let bufferData = Data(bytesNoCopy: mutableRawPointer,
                          count: count, deallocator: .none)

    var rgbBytes = [Float](repeating: 0, count: byteCount)
    var index = 0

    for component in bufferData.enumerated() {
        let offset = component.offset
        let isAlphaComponent = (offset % alphaComponent.baseOffset) ==
            alphaComponent.moduloRemainder

        guard !isAlphaComponent else { continue }
    }
}
```

```

        rgbBytes[index] = Float(component.element) / 255.0
        index += 1
    }

    return rgbBytes.withUnsafeBufferPointer(Data.init)
}

```

This code uses an extension called `Data` that copies the raw bytes into an array:

```

extension Data {
    init<T>(copyingBufferOf array: [T]) {
        self = array.withUnsafeBufferPointer(Data.init)
    }
}

```

Now you can pass the thumbnail pixel buffer you just created to `rgbDataFromBuffer`:

```

guard let rgbData = rgbDataFromBuffer(
    thumbnailPixelBuffer,
    byteCount: 224 * 224 * 3
)
else {
    print("Failed to convert the image buffer to RGB data.")
    return nil
}

```

At this point you have the raw RGB data that is in the format the model expects, and you can copy it directly to the input tensor:

```

try interpreter.allocateTensors()
try interpreter.copy(rgbData, toInputAt: 0)

```

You can then invoke the interpreter and read the output tensor:

```

try interpreter.invoke()
outputTensor = try interpreter.output(at: 0)

```

In the case of *Dogs vs. Cats*, you have as output a float array with two values, the first being the probability that the image is of a cat and the second that it's a dog. This is the same results code as you saw earlier, and it uses the same `Array` extension from the previous example:

```

let results = [Float32](unsafeData: outputTensor.data) ?? []

```

As you can see, although this is a more complex example, the same design pattern holds. You must understand your model's architecture, and the raw input and output formats. You must then structure your input data in the way the model expects—which often means getting down to raw bytes that you write into a buffer, or at least simulate using an array. You then have to read the raw stream of bytes coming out of the model and create a data structure to hold them. From the output perspective this will almost always be something like we've seen in this chapter—an array of floats. With the helper code you've implemented, you're most of the way there!

TensorFlow Lite Sample Apps

The TensorFlow team has built a large set of sample apps and is constantly adding to it. Armed with what you’ve learned in this chapter, you’ll be able to explore these and understand their input/output logic. At the time of writing, for iOS there are sample apps for:

Image classification

Read the device’s camera and classify up to a thousand different items.

Object detection

Read the device’s camera and give bounding boxes to objects that are detected.

Pose estimation

Take a look at the figures in the camera and infer their poses.

Speech recognition

Recognize common verbal commands.

Gesture recognition

Train a model for hand gestures and recognize them in the camera.

Image segmentation

Similar to object detection, but predict which class each pixel in an image belongs to.

Digit classifier

Recognize handwritten digits.

Summary

In this chapter you learned how to incorporate TensorFlow Lite into iOS apps by taking a comprehensive walkthrough of building a simple app that used the interpreter to invoke a model to perform inference. In particular, you saw how when dealing with models you have to get low-level with the data, ensuring that your input matches what the model expects. You also saw how to parse the raw data that comes out of the model. This is just the beginning of a long and fun journey toward putting machine learning into the hands of iOS users. In the next chapter we’ll move away from native mobile development to look at how TensorFlow.js can be used to train and run inference on models in the browser.

An Introduction to TensorFlow.js

In addition to TensorFlow Lite, which enables running on native mobile or embedded systems, the TensorFlow ecosystem also includes TensorFlow.js, which lets you develop ML models using the popular JavaScript language to use directly in the browser, or on a backend with Node.js. It allows you to train new models as well as running inference on them, and includes tools that let you convert your Python-based models into JavaScript-compatible ones. In this chapter you'll get an introduction to how TensorFlow.js fits into the overall ecosystem and a tour of its architecture, and you'll learn how to build your own models using a free, open source IDE that integrates with your browser.

What Is TensorFlow.js?

The TensorFlow ecosystem is summarized in [Figure 15-1](#). It comprises a suite of tools for *training* models, a repository for *preexisting* models and layers, and a set of technologies that allow you to *deploy* models for your end users to take advantage of.

Like TensorFlow Lite (Chapters [12–14](#)) and TensorFlow Serving ([Chapter 19](#)), TensorFlow.js *mostly* lives on the right side of this diagram, because while it's primarily intended as a runtime for models, it can also be used for training models and should be considered a first-class language alongside Python and Swift for this task. TensorFlow.js can be run in the browser or on backends like Node.js, but for the purposes of this book we'll focus primarily on the browser.

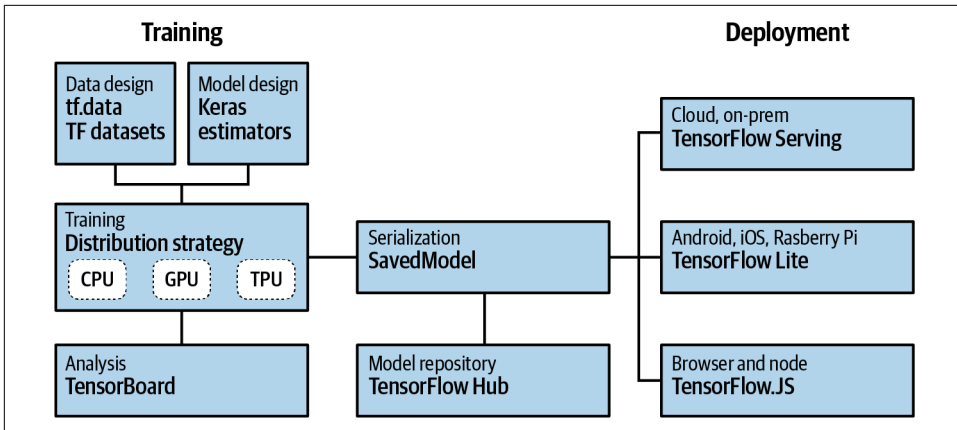


Figure 15-1. The TensorFlow ecosystem

The architecture of how TensorFlow.js gives you browser-based training and inference is shown in [Figure 15-2](#).

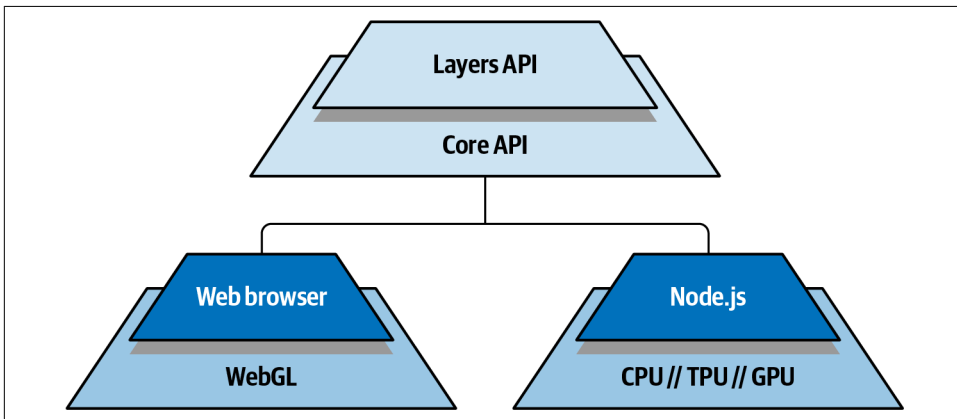


Figure 15-2. TensorFlow.js high-level architecture

As a developer, you'll typically use the Layers API, which gives Keras-like syntax in JavaScript, allowing you to use the skills you learned at the beginning of this book in JavaScript. This is underpinned by the Core API, which gives, as its name suggests, the core TensorFlow functionality in JavaScript. As well as providing the basis for the Layers API, it allows you to reuse existing Python-based models by means of a conversion toolkit that puts them into a JSON-based format for easy consumption.

The Core API then can run in a web browser, taking advantage of GPU-based acceleration using WebGL, or on Node.js, where, depending on the configuration of the environment, it can take advantage of TPU- or GPU-based acceleration in addition to the CPU.

If you're not used to web development in either HTML or JavaScript, don't worry; this chapter will serve as a primer, giving you enough background to help you build your first models. While you can use any web/JavaScript development environment you like, I would recommend one called **Brackets** to new users. In the next section you'll see how to install that and get it up and running, after which you'll build your first model.

Installing and Using the Brackets IDE

Brackets is a free, open source text editor that is extremely useful for web developers—in particular new ones—in that it integrates neatly with the browser, allowing you to serve your files locally so you can test and debug them. Often, when setting up web development environments, that's the tricky part. It's easy to write HTML or JavaScript code, but without a server to serve them to your browser, it's hard to really test and debug them. Brackets is available for Windows, Mac, and Linux, so whatever operating system you're using, the experience should be similar. For this chapter, I tried it out on Mint Linux, and it worked really well!

After you download and install Brackets, run it and you'll see the Getting Started page, similar to **Figure 15-3**. In the top-right corner you'll see a lightning bolt icon.

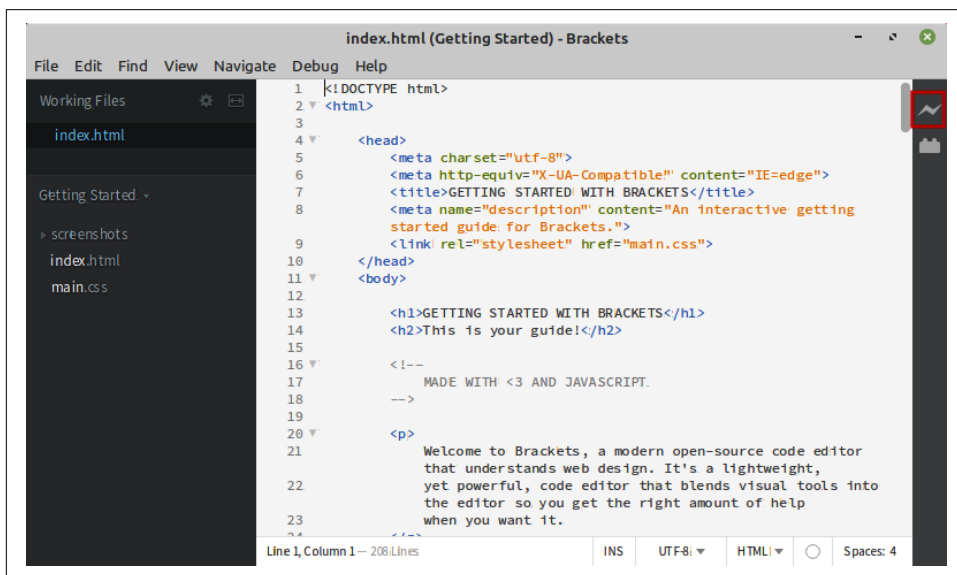


Figure 15-3. Brackets welcome screen

Click that and your web browser will launch. As you edit the HTML code in Brackets, the browser will live update. So, for example, if you change the code on line 13 that says:


```
<h1>GETTING STARTED WITH BRACKETS</h1>
```

to something else, such as:

```
<h1>Hello, TensorFlow Readers!</h1>
```

You'll see the contents in the browser change in real time to match your edits, as shown in [Figure 15-4](#).

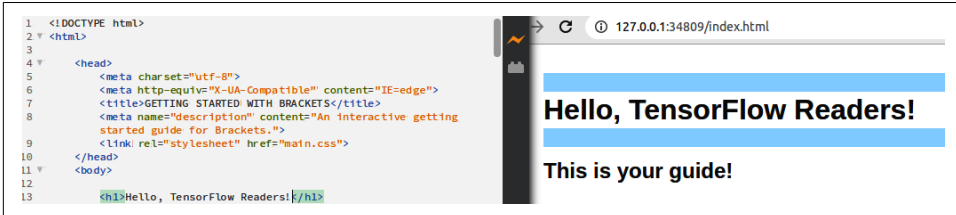


Figure 15-4. Real-time updates in the browser

I find this really handy for HTML and JavaScript development in the browser, because the environment just gets out of the way and lets you focus on your code. With so many new concepts, particularly in machine learning, this is invaluable because it helps you work without too many distractions.

You'll notice, on the Getting Started page, that you're working in just a plain directory that Brackets serves files from. If you want to use your own, just create a directory in your filesystem and open that. New files you create in Brackets will be created and run from there. Make sure it's a directory you have write access to so you can save your work!

Now that you have a development environment up and running, it's time to create your first machine learning model in JavaScript. For this we'll go back to our "Hello World" scenario where you train a model that infers the relationship between two numbers. If you've been working through this book from the beginning, you've seen this model many times already, but it's still a useful one for helping you understand the syntactical differences you'll need to consider when programming in JavaScript!

Building Your First TensorFlow.js Model

Before using TensorFlow.js in the browser, you'll need to host the JavaScript in an HTML file. Create one and populate it with this skeleton HTML:

```
<html>
<head></head>
<body>
  <h1>First HTML Page</h1>
</body>
</html>
```

Then, beneath the `<head>` section and before the `<body>` tag, you can insert a `<script>` tag specifying the location of the TensorFlow.js library:

```
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"></script>
```

If you were to run the page now, TensorFlow.js would be downloaded, but you wouldn't see any impact.

Next, add another `<script>` tag immediately below the first one. Within it, you can create a model definition. Note that while it's very similar to how you would do it in TensorFlow in Python (refer back to [Chapter 1](#) for details), there are some differences. For example, in JavaScript every line ends with a semicolon. Also, parameters to functions such as `model.add` or `model.compile` are in JSON notation.

This model is the familiar “Hello World” one, consisting of a single layer with a single neuron. It will be compiled with mean squared error as the loss function and stochastic gradient descent as the optimizer:

```
<script lang="js">
  const model = tf.sequential();
  model.add(tf.layers.dense({units: 1, inputShape: [1]}));
  model.compile({loss: 'meanSquaredError', optimizer: 'sgd'});
```

Next, you can add the data. This is a little different from Python, where you had Numpy arrays. Of course, these aren't available in JavaScript, so you'll use the `tf.tensor2d` structure instead. It's close, but there's one key difference:

```
const xs = tf.tensor2d([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], [6, 1]);
const ys = tf.tensor2d([-3.0, -1.0, 2.0, 3.0, 5.0, 7.0], [6, 1]);
```

Note that, as well as the list of values, you also have a second array that defines the *shape* of the first one. So, your `tensor2d` is initialized with a 6×1 list of values, followed by an array containing `[6,1]`. If you were to feed seven values in, the second parameter would be `[7,1]`.

To do the training, you can then create a function called `doTraining`. This will train the model using `model.fit`, and, as before, the parameters to it will be formatted as a JSON list:

```
async function doTraining(model){
  const history =
    await model.fit(xs, ys,
      { epochs: 500,
        callbacks:{
          onEpochEnd: async(epoch, logs) =>{
            console.log("Epoch:"
              + epoch
              + " Loss:"
              + logs.loss);
          }
        }
      }
    )
}
```

```

    });
}

```

It's an asynchronous operation—the training will take a period of time—so it's best to create this as an async function. Then you `await` the `model.fit`, passing it the number of epochs as a parameter. You can also specify a callback that will write out the loss for each epoch when it ends.

The last thing to do is call this `doTraining` method, passing it the model and reporting on the result after it finishes training:

```

doTraining(model).then(() => {
  alert(model.predict(tf.tensor2d([10], [1,1])));
});

```

This calls `model.predict`, passing it a single value to get a prediction from. Because it is using a `tensor2d` as well as the value to predict, you also have to pass a second parameter with the shape of the first. So to predict the result for 10, you create a `tensor2d` with this value in an array and then pass in the shape of that array.

For convenience, here's the complete code:

```

<html>
<head></head>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"></script>
<script lang="js">
  async function doTraining(model){
    const history =
      await model.fit(xs, ys,
        { epochs: 500,
          callbacks:{
            onEpochEnd: async(epoch, logs) =>{
              console.log("Epoch:"
                + epoch
                + " Loss:"
                + logs.loss);
            }
          }
        });
  }

  const model = tf.sequential();
  model.add(tf.layers.dense({units: 1, inputShape: [1]}));
  model.compile({loss: 'meanSquaredError',
    optimizer: 'sgd'});
  model.summary();
  const xs = tf.tensor2d([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], [6, 1]);
  const ys = tf.tensor2d([-3.0, -1.0, 2.0, 3.0, 5.0, 7.0], [6, 1]);
  doTraining(model).then(() => {
    alert(model.predict(tf.tensor2d([10], [1,1])));
  });
</script>

```

```

<body>
  <h1>First HTML Page</h1>
</body>
</html>

```

When you run this page, it will appear as if nothing has happened. Wait a few seconds, and then a dialog will appear like the one in [Figure 15-5](#). This is the alert dialog that is shown with the results of the prediction for [10].

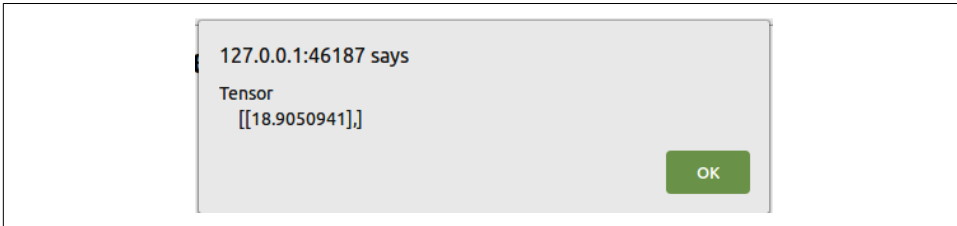


Figure 15-5. Results of the inference after training

It can be a little disconcerting to have that long pause before the dialog is shown—as you’ve probably guessed, the model was training during that period. Recall that in the `doTraining` function, you created a callback that writes the loss per epoch to the console log. If you want to see this, you can do so with the browser’s developer tools. In Chrome you can access these by clicking the three dots in the upper-right corner and selecting **More Tools → Developer Tools**, or by pressing **Ctrl-Shift-I**.

Once you have them, select **Console** at the top of the pane and refresh the page. As the model retrains, you’ll see the loss per epoch (see [Figure 15-6](#)).

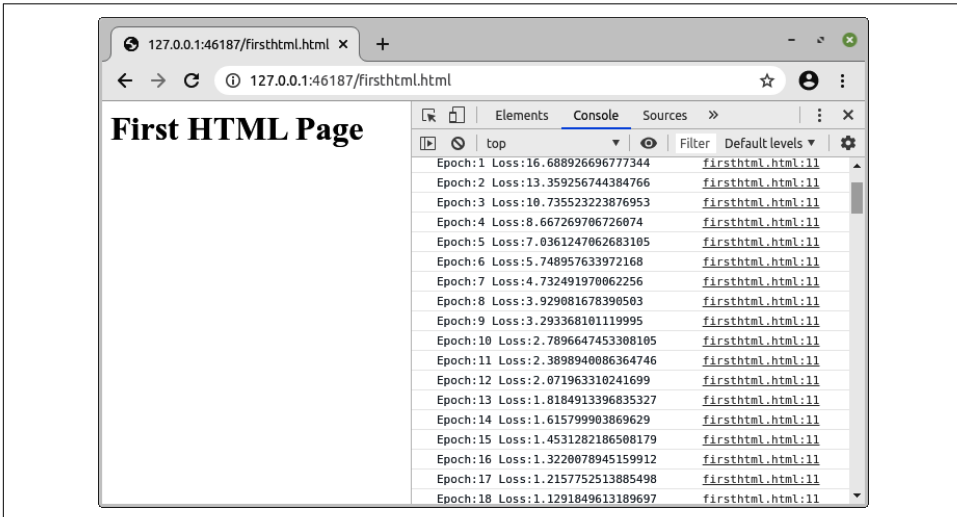


Figure 15-6. Exploring the per-epoch loss in the browser’s developer tools

Now that you've gone through your first (and simplest) model, you're ready to build something a little more complex.

Creating an Iris Classifier

The last example was a very simple one, so let's work on one that's a little more complex next. If you've done any work with machine learning, you've probably heard about the Iris dataset, which is a perfect one for learning ML.

The dataset contains 150 data items with four attributes each, describing three classes of flower. The attributes are sepal length and width, and petal length and width. When plotted against each other, clear clusters of flower types are seen (Figure 15-7).

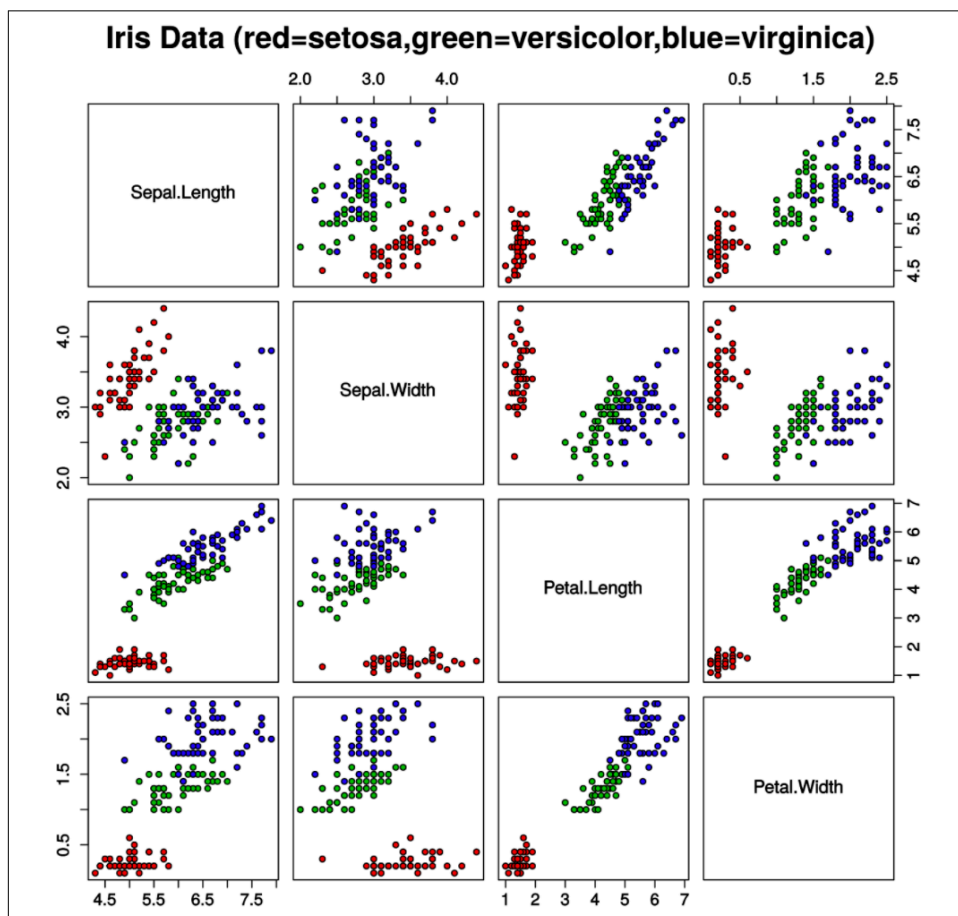


Figure 15-7. Plotting features in the Iris dataset (source: Nicoguaro, available on [Wiki-media Commons](#))

Figure 15-7 shows the complexity of the problem, even with a simple dataset like this. How does one separate the three types of flowers using rules? The petal length versus petal width plots come close, with the *Iris setosa* samples (in red) being very distinct from the others, but the blue and green sets are intertwined. This makes for an ideal learning set in ML: it's small, so fast to train, and you can use it to solve a problem that's difficult to do in rules-based programming!

You can download the dataset from the [UCI Machine Learning Repository](#), or use the version in the book's [GitHub repository](#), which I've converted to CSV to make it easier to use in JavaScript.

The CSV looks like this:

```
sepal_length,sepal_width,petal_length,petal_width,species
5.1,3.5,1.4,0.2,setosa
4.9,3,1.4,0.2,setosa
4.7,3.2,1.3,0.2,setosa
4.6,3.1,1.5,0.2,setosa
5.3,6,1.4,0.2,setosa
5.4,3.9,1.7,0.4,setosa
4.6,3.4,1.4,0.3,setosa
5.3,4,1.5,0.2,setosa
...
```

The four data points for each flower are the first four values. The label is the fifth value, one of *setosa*, *versicolor*, or *virginica*. The first line in the CSV file contains the column labels. Keep that in mind—it will be useful later!

To get started, create a basic HTML page as before, and add the `<script>` tag to load TensorFlow.js:

```
<html>
<head></head>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"></script>
<body>
  <h1>Iris Classifier</h1>
</body>
</html>
```

To load a CSV file, TensorFlow.js has the `tf.data.csv` API that you can give a URL to. This also allows you to specify which column is the label. Because the first line in the CSV file I've prepared contains column names, you can specify which one contains the labels, which in this case is `species`, as follows:

```
<script lang="js">
  async function run(){
    const csvUrl = 'iris.csv';
    const trainingData = tf.data.csv(csvUrl, {
      columnConfigs: {
        species: {
          isLabel: true
        }
      }
    });
  }
}
```

```
    }
  });
}
```

The labels are strings, which you don't really want to train a neural network with. This is going to be a multiclass classifier with three output neurons, each containing the probability that the input data represents the respective species of flower. Thus, a one-hot encoding of the labels is perfect.

This way, if you represent *setosa* as [1, 0, 0], showing that you want the first neuron to light up for this class, *virginica* as [0, 1, 0], and *versicolor* as [0, 0, 1], you're effectively defining the template for how the final layer neurons should behave for each class.

Because you used `tf.data` to load the data, you have the ability to use a mapping function that will take in your `xs` (features) and `ys` (labels) and map them differently. So, to keep the features intact and one-hot encode the labels, you can write code like this:

```
const convertedData =
  trainingData.map(({xs, ys}) => {
    const labels = [
      ys.species == "setosa" ? 1 : 0,
      ys.species == "virginica" ? 1 : 0,
      ys.species == "versicolor" ? 1 : 0
    ]
    return{ xs: Object.values(xs), ys: Object.values(labels)};
  }).batch(10);
```

Note that the labels are stored as an array of three values. Each value defaults to 0, unless the species matches the given string, in which case it will be a 1. Thus, *setosa* will be encoded as [1, 0, 0], and so on.

The mapping function will return the `xs` unaltered and the `ys` one-hot encoded.

You can now define your model. Your input layer's shape is the number of features, which is the number of columns in the CSV file minus 1 (because one of the columns represents the labels):

```
const numOfFeatures = (await trainingData.columnNames()).length - 1;

const model = tf.sequential();
model.add(tf.layers.dense({inputShape: [numOfFeatures],
  activation: "sigmoid", units: 5}))

model.add(tf.layers.dense({activation: "softmax", units: 3}));
```

Your final layer has three units because of the three classes that are one-hot encoded in the training data.

Next, you'll specify the loss function and optimizer. Because this is a multiple-category classifier, make sure you use a categorical loss function such as categorical cross entropy. You can use an optimizer such as adam in the `tf.train` namespace and pass it parameters like the learning rate (here, `0.06`):

```
model.compile({loss: "categoricalCrossentropy",
               optimizer: tf.train.adam(0.06)});
```

Because the data was formatted as a dataset, you can use `model.fitDataset` for training instead of `model.fit`. To train for one hundred epochs and catch the loss in the console, you can use a callback like this:

```
await model.fitDataset(convertedData,
                      {epochs:100,
                      callbacks:{
                        onEpochEnd: async(epoch, logs) =>{
                          console.log("Epoch: " + epoch +
                                      " Loss: " + logs.loss);
                        }
                      }});
```

To test the model after it has finished training, you can load values into a `tensor2d`. Don't forget that when using a `tensor2d`, you also have to specify the shape of the data. In this case, to test a set of four values, you would define them in a `tensor2d` like this:

```
const testVal = tf.tensor2d([4.4, 2.9, 1.4, 0.2], [1, 4]);
```

You can then get a prediction by passing this to `model.predict`:

```
const prediction = model.predict(testVal);
```

You'll get a tensor value back that looks something like this:

```
[[0.9968228, 0.00000029, 0.0031742],]
```

To get the largest value, you can use the `argMax` function:

```
tf.argmax(prediction, axis=1)
```

This will return `[0]` for the preceding data because the neuron at position 0 had the highest probability.

To unpack this into a value, you can use `.dataSync`. This operation synchronously downloads a value from the tensor. It does block the UI thread, so be careful when using it!

The following code will simply return `0` instead of `[0]`:

```
const pIndex = tf.argmax(prediction, axis=1).dataSync();
```


To map this back to a string with the class name, you can then use this code:

```
const classNames = ["Setosa", "Virginica", "Versicolor"];  
alert(classNames[pIndex])
```

Now you've seen how to load data from a CSV file, turn it into a dataset, and then fit a model from that dataset, as well as how to run predictions from that model. You're well equipped to experiment with other datasets of your choosing to further hone your skills!

Summary

This chapter introduced you to TensorFlow.js and how it can be used to train models and perform inference in the browser. You saw how to use the open source Brackets IDE to code and test your models on a local web server, and used this to train your first two models: a “Hello World” linear regression model and a basic classifier for the popular Iris dataset. These were very simple scenarios, but in [Chapter 16](#) you'll take things to the next level and see how to train computer vision models with TensorFlow.js.

Coding Techniques for Computer Vision in TensorFlow.js

In Chapters 2 and 3 you saw how TensorFlow can be used to create models for computer vision, which can be trained to recognize the content in images. In this chapter you'll do the same, but with JavaScript. You'll build a handwriting recognizer that runs in the browser and is trained on the MNIST dataset. You can see it in Figure 16-1.

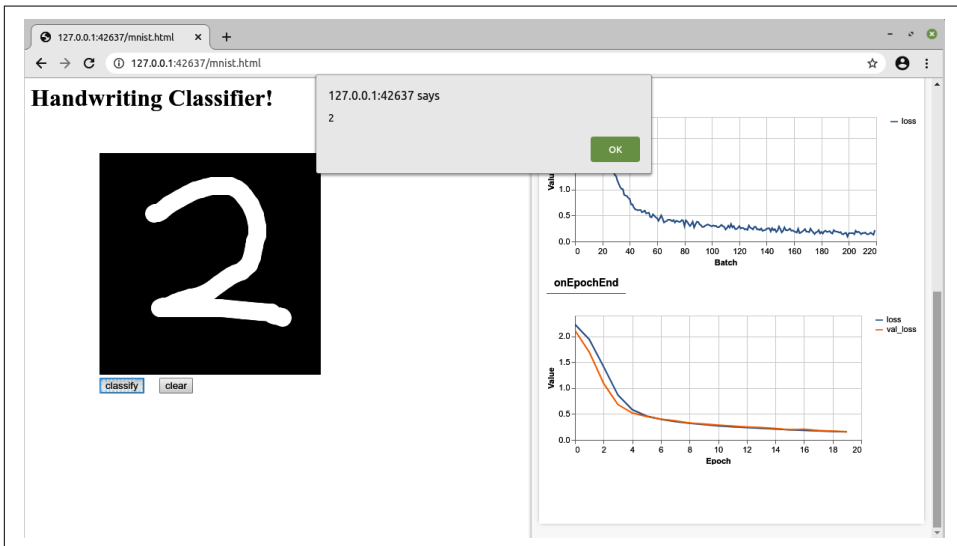


Figure 16-1. A handwriting classifier in the browser

There are a few crucial implementation details to be aware of when you're working with TensorFlow.js, particularly if you are building applications in the browser. Perhaps the biggest and most important of these is how training data is handled. When using a browser, every time you open a resource at a URL, you're making an HTTP connection. You use this connection to pass commands to a server, which will then dispatch the results for you to parse. When it comes to machine learning, you generally have a lot of training data—for example, in the case of MNIST and Fashion MNIST, even though they are small learning datasets they still each contain 70,000 images, which would be 70,000 HTTP connections! You'll see how to deal with this later in this chapter.

Additionally, as you saw in the last chapter, even for a very simple scenario like training for $Y = 2X - 1$, nothing appeared to happen during the training cycle unless you opened the debug console, where you could see the epoch-by-epoch loss. If you're training something much more sophisticated, which takes longer, it can be difficult to understand what's going on during training. Fortunately there are built-in visualization tools that you can use, as seen on the right side of [Figure 16-1](#); you'll also explore them in this chapter.

There are also syntactical differences to be aware of when defining a convolutional neural network in JavaScript, some of which we touched on in the previous chapter. We'll start by considering these. If you need a refresher on CNNs, see [Chapter 3](#).

JavaScript Considerations for TensorFlow Developers

When building a full (or close to it) application in JavaScript like you will in this chapter, there are a number of things that you'll have to take into account. JavaScript is very different from Python, and, as such, while the TensorFlow.js team has worked hard to keep the experience as close to “traditional” TensorFlow as possible, there are some changes.

First is the *syntax*. While in many respects TensorFlow code in JavaScript (especially Keras code) is quite similar to that in Python, there are a few syntactic differences—most notably, as mentioned in the previous chapter, the use of JSON in parameter lists.

Next is *synchronicity*. Especially when running in the browser, you can't lock up the UI thread when training and instead need to perform many operations asynchronously, using JavaScript Promises and `await` calls. It's not the intention of this chapter to go into depth teaching these concepts; if you aren't already familiar with them, you can think of them as asynchronous functions that, instead of waiting to finish executing before returning, will go off and do their own thing and “call you back” when they're done. The `tfjs-vis` library was created to help you debug your code when training models asynchronously with TensorFlow.js. The visualization tools give you

a separate sidebar in the browser, not interfering with your current page, in which visualizations like training progress can be plotted; we'll talk more about them in [“Using Callbacks for Visualization” on page 279](#).

Resource usage is also an important consideration. As the browser is a shared environment, you may have multiple tabs open in which you're doing different things, or you might be performing multiple operations within the same web app. Therefore, it's important to control how much memory you use. ML training can be memory-intensive, as lots of data is required to understand and distinguish the patterns that map features to labels. As a result, you should take care to tidy up after yourself. The `tidy` API is designed for just that and should be used as much as possible: wrapping a function in `tidy` ensures that all tensors not returned by the function will be cleaned up and released from memory.

While not a TensorFlow API, the `arrayBuffer` in JavaScript is another handy construct. It's analogous to a `ByteBuffer` for managing data like it was low-level memory. In the case of machine learning applications, it's often easiest to use very sparse encoding, as you've seen already with one-hot encoding. Remembering that processing in JavaScript can be thread-intensive and you don't want to lock up the browser, it can be easier to have a sparse encoding of data that doesn't require processor power to decode. In the example from this chapter, the labels are encoded in this way: for each of the 10 classes, 9 of them will have a 0×00 byte and the other, representing the matching class for that feature, will have a 0×01 byte. This means 10 bytes, or 80 bits, are used for each label, where as a coder you might think that only 4 bits would be necessary to encode a number between 1 and 10. But of course, if you did it that way you would have to decode the results—65,000 times for that many labels. Thus, having a sparsely encoded file that's easily represented in bytes by an `arrayBuffer` can be quicker, albeit with a larger file size.

Also worthy of mention are the `tf.browser` APIs, which are helpful for dealing with images. At the time of writing there are two methods, `tf.browser.toPixels` and `tf.browser.fromPixels`, which, as their names suggest, are used for translating pixels between browser-friendly formats and tensor formats. You'll use these later when you want to draw a picture and have it interpreted by the model.

Building a CNN in JavaScript

When building any neural network with TensorFlow Keras, you define a number of layers. In the case of a convolutional neural network, you'll typically have a series of convolutional layers followed by pooling layers, whose output is flattened and fed into a dense layer. For example, here's an example of a CNN that was defined for classifying the MNIST dataset back in [Chapter 3](#):

```

model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu',
                           input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])

```

Let's break down line by line how this could be implemented in JavaScript. We'll start by defining the model as a sequential:

```
model = tf.sequential();
```

Next, we'll define the first layer as a 2D convolution that learns 64 filters, with a kernel size of 3×3 and an input shape of $28 \times 28 \times 1$. The syntax here is very different from Python, but you can see the similarity:

```
model.add(tf.layers.conv2d({inputShape: [28, 28, 1],
                                   kernelSize: 3, filters: 64, activation: 'relu'}));
```

The following layer was a MaxPooling2D, with a pool size of 2×2 . In JavaScript it's implemented like this:

```
model.add(tf.layers.maxPooling2d({poolSize: [2, 2]}));
```

This was followed by another convolutional layer and max pooling layer. The difference here is that there is no input shape, as it isn't an input layer. In JavaScript this looks like this:

```
model.add(tf.layers.conv2d({filters: 64,
                                   kernelSize: 3, activation: 'relu'}));
```

```
model.add(tf.layers.maxPooling2d({poolSize: [2, 2]}));
```

After this, the output was flattened, and in JavaScript the syntax for that is:

```
model.add(tf.layers.flatten());
```

The model was then completed by two dense layers, one with 128 neurons activated by relu, and the output layer of 10 neurons activated by softmax:

```
model.add(tf.layers.dense({units: 128, activation: 'relu'}));
```

```
model.add(tf.layers.dense({units: 10, activation: 'softmax'}));
```

As you can see, the JavaScript APIs look very similar to the Python ones, but there are syntactical differences that can be gotchas: the names of APIs follow camel case convention but start with a lowercase letter, as expected in JavaScript (i.e., `maxPooling2D` instead of `MaxPooling2D`), parameters are defined in JSON instead of comma-separated lists, etc. Keep an eye on these differences as you code your neural networks in JavaScript.

For convenience, here's the complete JavaScript definition of the model:

```
model = tf.sequential();

model.add(tf.layers.conv2d({inputShape: [28, 28, 1],
                                   kernelSize: 3, filters: 8, activation: 'relu'}));

model.add(tf.layers.maxPooling2d({poolSize: [2, 2]}));

model.add(tf.layers.conv2d({filters: 16,
                                   kernelSize: 3, activation: 'relu'}));

model.add(tf.layers.maxPooling2d({poolSize: [2, 2]}));

model.add(tf.layers.flatten());

model.add(tf.layers.dense({units: 128, activation: 'relu'}));

model.add(tf.layers.dense({units: 10, activation: 'softmax'}));
```

Similarly, when compiling the model, consider the differences between Python and JavaScript. Here's the Python:

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

And the equivalent JavaScript:

```
model.compile(
{  optimizer: tf.train.adam(),
   loss: 'categoricalCrossentropy',
   metrics: ['accuracy']
});
```

While they're very similar, keep in mind the JSON syntax for the parameters (*parameter: value*, not *parameter=value*) and that the list of parameters is enclosed in curly braces (`{}`).

Using Callbacks for Visualization

In [Chapter 15](#), when you were training the simple neural network, you logged the loss to the console when each epoch ended. You then used the browser's developer tools to view the progress in the console, looking at the changes in the loss over time. A more sophisticated approach is to use the [TensorFlow.js visualization tools](#), created specifically for in-browser development. These include tools for reporting on training metrics, model evaluation, and more. The visualization tools appear in a separate area of the browser window that doesn't interfere with the rest of your web page. The term used for this is a *visor*. It will default to showing at the very least the model architecture.

To use the `tfjs-vis` library in your page, you can include it with a script:

```
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs-vis"></script>
```

Then, to see visualizations while training, you need to specify a callback in your `model.fit` call. Here's an example:

```
return model.fit(trainXs, trainYs, {  
  batchSize: BATCH_SIZE,  
  validationData: [testXs, testYs],  
  epochs: 20,  
  shuffle: true,  
  callbacks: fitCallbacks  
});
```

The callbacks are defined as a `const`, using `tfvis.show.fitCallbacks`. This takes two parameters—a container and the desired metrics. These are also defined using `consts`, as shown here:

```
const metrics = ['loss', 'val_loss', 'accuracy', 'val_accuracy'];  
  
const container = { name: 'Model Training', styles: { height: '640px' },  
  tab: 'Training Progress' };  
  
const fitCallbacks = tfvis.show.fitCallbacks(container, metrics);
```

The container `const` has parameters that define the visualization area. All visualizations are shown in a single tab by default. By using a `tab` parameter (set to “Training Progress” here), you can split the training progress out into a separate tab. [Figure 16-2](#) illustrates what the preceding code will show in the visualization area at runtime.

Next, let's explore how to manage the training data. As mentioned earlier, handling thousands of images through URL connections is bad for the browser because it will lock up the UI thread. But there are some tricks that you can use from the world of game development!

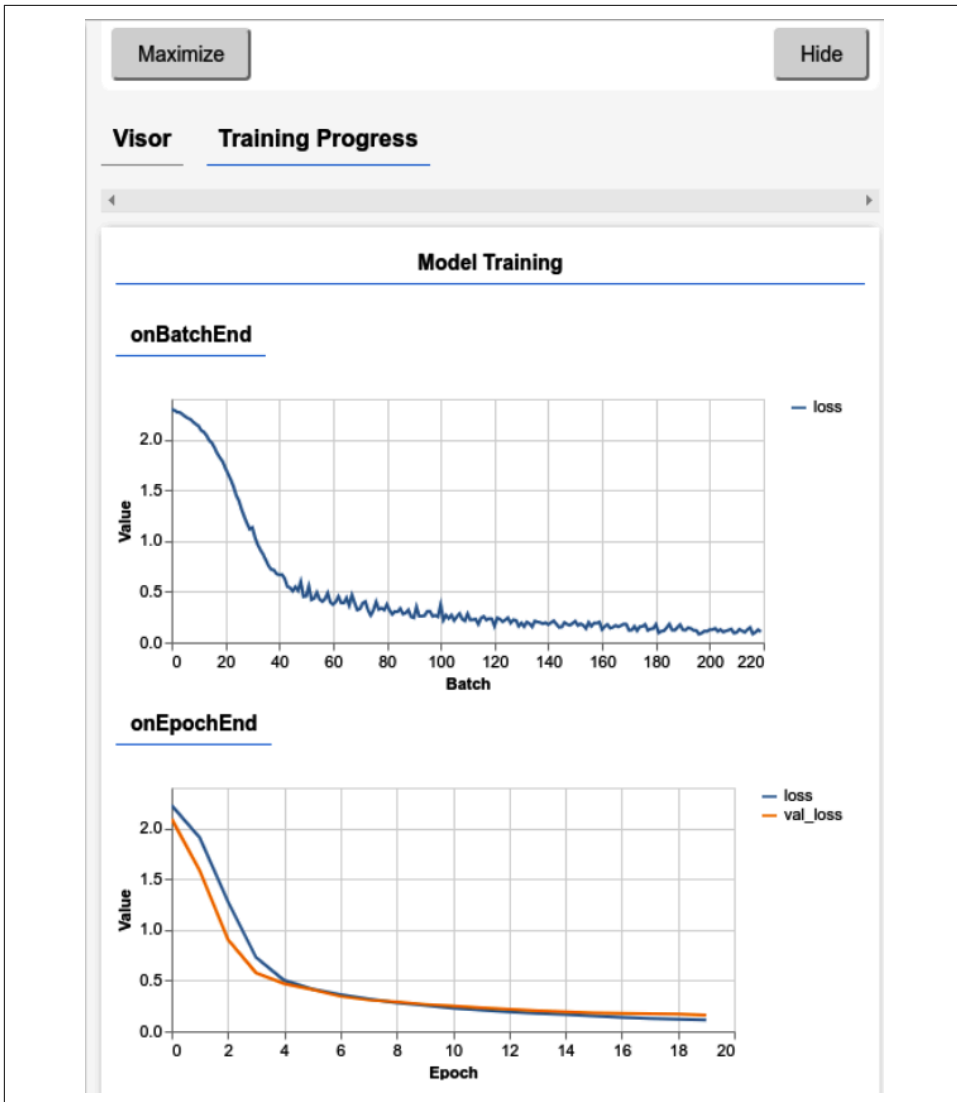


Figure 16-2. Using the visualization tools

Training with the MNIST Dataset

Instead of downloading every image one by one, a useful way to handle training of data in TensorFlow.js is to append all the images together into a single image, often called a *sprite sheet*. This technique is commonly used in game development, where the graphics of a game are stored in a single file instead of multiple smaller ones for file storage efficiency. If we were to store all the images for training in a single file, we'd just need to open one HTTP connection to it in order to download them all in a single shot.

For the purposes of learning, the TensorFlow team has created sprite sheets from the MNIST and Fashion MNIST datasets that we can use here. For example, the MNIST images are available in a file called *mnist_images.png* (see Figure 16-3).

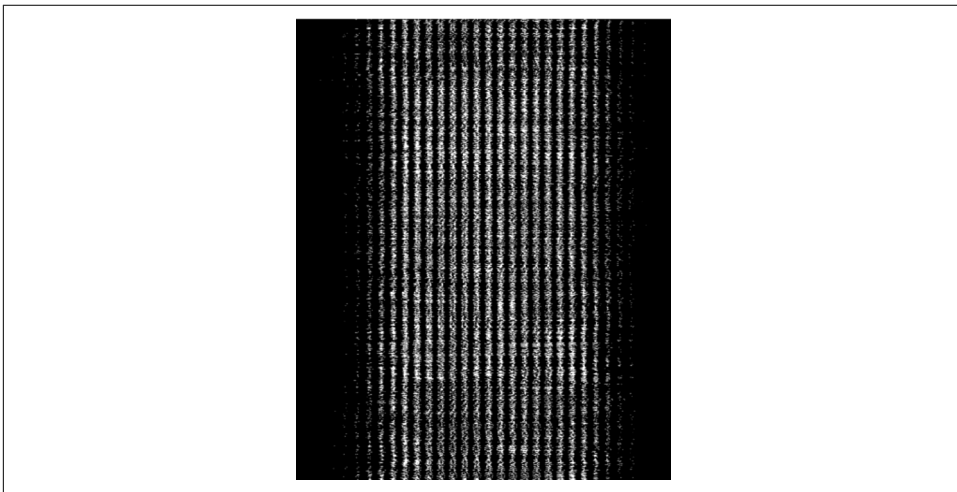


Figure 16-3. An excerpt from *mnist_images.png* in an image viewer

If you explore the dimensions of this image, you'll see that it has 65,000 lines, each with 784 (28×28) pixels in it. If those dimensions look familiar, you might recall that MNIST images are 28×28 monochrome. So, you can download this image, read it line by line, and then take each of the lines and separate it into a 28×28 -pixel image.

You can do this in JavaScript by loading the image, and then defining a canvas on which you can draw the individual lines after extracting them from the original image. The bytes from these canvases can then be extracted into a dataset that you'll use for training. This might seem a bit convoluted, but given that JavaScript is an in-browser technology, it wasn't really designed for data and image processing like this. That said, it works really well, and runs really quickly! Before we get into the details of that, however, you should also look at the labels and how they're stored.

First, set up constants for the training and test data, bearing in mind that the MNIST image has 65,000 lines, one for each image. The ratio of training to testing data can be defined as 5:1, and from this you can calculate the number of elements for training and the number for testing:

```
const IMAGE_SIZE = 784;
const NUM_CLASSES = 10;
const NUM_DATASET_ELEMENTS = 65000;

const TRAIN_TEST_RATIO = 5 / 6;

const NUM_TRAIN_ELEMENTS = Math.floor(TRAIN_TEST_RATIO * NUM_DATASET_ELEMENTS);
const NUM_TEST_ELEMENTS = NUM_DATASET_ELEMENTS - NUM_TRAIN_ELEMENTS;
```

Note that all of this code is in the [repo](#) for this book, so please feel free to adapt it from there!

Next up, you need to create some constants for the image control that will hold the sprite sheet and the canvas that can be used for slicing it up:

```
const img = new Image();
const canvas = document.createElement('canvas');
const ctx = canvas.getContext('2d');
```

To load the image, you simply set the `img` control to the path of the sprite sheet:

```
img.src = MNIST_IMAGES_SPRITE_PATH;
```

Once the image is loaded, you can set up a buffer to hold the bytes in it. The image is a PNG file, which has 4 bytes per pixel, so you'll need to reserve 65,000 (number of images) \times 768 (number of pixels in a 28×28 image) \times 4 (number of bytes in a PNG per pixel) bytes for the buffer. You don't need to split the file image by image, but can split it in chunks. Take five thousand images at a time by specifying the `chunkSize` as shown here:

```
img.onload = () => {
  img.width = img.naturalWidth;
  img.height = img.naturalHeight;

  const datasetBytesBuffer =
    new ArrayBuffer(NUM_DATASET_ELEMENTS * IMAGE_SIZE * 4);

  const chunkSize = 5000;
  canvas.width = img.width;
  canvas.height = chunkSize;
```

Now you can create a loop to go through the image in chunks, creating a set of bytes for each chunk and drawing it to the canvas. This will decode the PNG into the canvas, giving you the ability to get the raw bytes from the image. As the individual images in the dataset are monochrome, the PNG will have the same levels for the R, G, and B bytes, so you can just take any of them:

```
for (let i = 0; i < NUM_DATASET_ELEMENTS / chunkSize; i++) {
  const datasetBytesView = new Float32Array(
    datasetBytesBuffer, i * IMAGE_SIZE * chunkSize * 4,
    IMAGE_SIZE * chunkSize);
  ctx.drawImage(
    img, 0, i * chunkSize, img.width, chunkSize, 0, 0, img.width,
    chunkSize);

  const imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);

  for (let j = 0; j < imageData.data.length / 4; j++) {
    // All channels hold an equal value since the image is grayscale, so
    // just read the red channel.
    datasetBytesView[j] = imageData.data[j * 4] / 255;
  }
}
```

The images can now be loaded into a dataset with:

```
this.datasetImages = new Float32Array(datasetBytesBuffer);
```

Similar to the images, the labels are stored in **a single file**. This is a binary file with a sparse encoding of the labels. Each label is represented by 10 bytes, with one of those bytes having the value 01 to represent the class. This is easier to understand with a visualization, so take a look at **Figure 16-4**.

This shows a hex view of the file with the first 10 bytes highlighted. Here, byte 8 is 01, while the rest are all 00. This indicates that the label for the first image is 8. Given that MNIST has 10 classes, for the digits 0 through 9, we know that the eighth label is for the number 7.

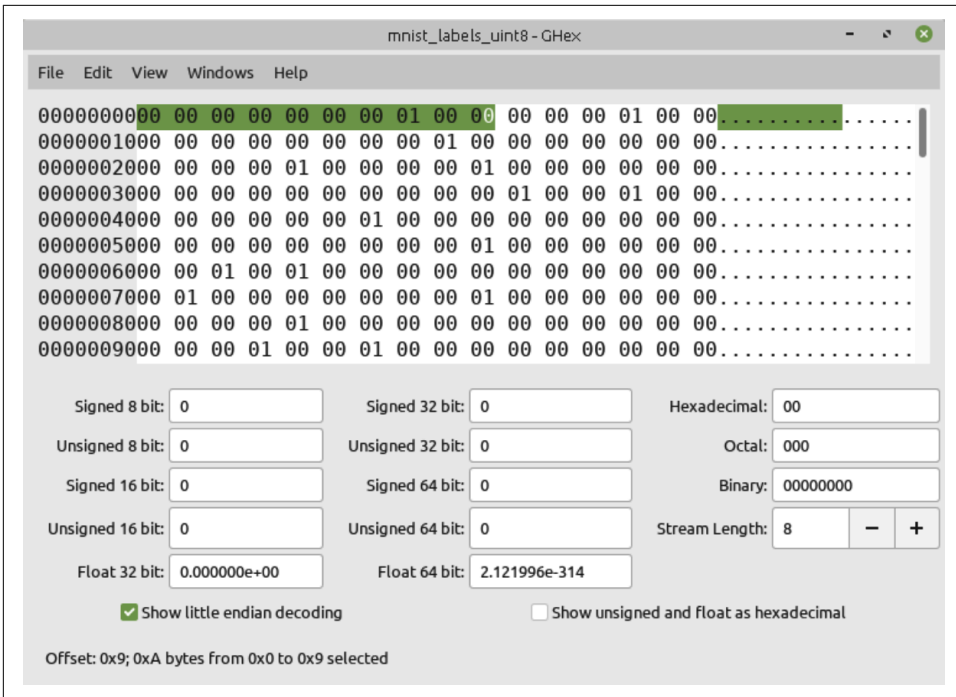


Figure 16-4. Exploring the labels file

So, as well as downloading and decoding the bytes for the images line by line, you'll also need to decode the labels. You download these alongside the image by fetching the URL, and then decode the labels into integer arrays using `arrayBuffer`:

```
const labelsRequest = fetch(MNIST_LABELS_PATH);
const [imgResponse, labelsResponse] =
  await Promise.all([imgRequest, labelsRequest]);

this.datasetLabels = new Uint8Array(await labelsResponse.arrayBuffer());
```

The sparseness of the encoding of the labels greatly simplifies the code—with this one line you can get all the labels into a buffer. If you were wondering why such an inefficient storage method was used for the labels, that was the trade-off: more complex storage but simpler decoding!

The images and labels can then be split into training and test sets:

```
this.trainImages =  
  this.datasetImages.slice(0, IMAGE_SIZE * NUM_TRAIN_ELEMENTS);  
this.testImages = this.datasetImages.slice(IMAGE_SIZE * NUM_TRAIN_ELEMENTS);  
  
this.trainLabels =  
  this.datasetLabels.slice(0, NUM_CLASSES * NUM_TRAIN_ELEMENTS);  
this.testLabels =  
  this.datasetLabels.slice(NUM_CLASSES * NUM_TRAIN_ELEMENTS);
```

For training, the data can also be batched. The images will be in `Float32Arrays` and the labels in `Uint8Arrays`. They're then converted into `tensor2d` types called `xs` and `labels`:

```
nextBatch(batchSize, data, index) {  
  const batchImagesArray = new Float32Array(batchSize * IMAGE_SIZE);  
  const batchLabelsArray = new Uint8Array(batchSize * NUM_CLASSES);  
  
  for (let i = 0; i < batchSize; i++) {  
    const idx = index();  
  
    const image =  
      data[0].slice(idx * IMAGE_SIZE, idx * IMAGE_SIZE + IMAGE_SIZE);  
    batchImagesArray.set(image, i * IMAGE_SIZE);  
  
    const label =  
      data[1].slice(idx * NUM_CLASSES, idx * NUM_CLASSES + NUM_CLASSES);  
    batchLabelsArray.set(label, i * NUM_CLASSES);  
  }  
  
  const xs = tf.tensor2d(batchImagesArray, [batchSize, IMAGE_SIZE]);  
  const labels = tf.tensor2d(batchLabelsArray, [batchSize, NUM_CLASSES]);  
  
  return {xs, labels};  
}
```

The training data can then use this batch function to return shuffled training batches of the desired batch size:

```
nextTrainBatch(batchSize) {  
  return this.nextBatch(  
    batchSize, [this.trainImages, this.trainLabels], () => {  
      this.shuffledTrainIndex =  
        (this.shuffledTrainIndex + 1) % this.trainIndices.length;  
      return this.trainIndices[this.shuffledTrainIndex];  
    });  
}
```

Test data can be batched and shuffled in exactly the same way.

Now, to get ready for training, you can set up some parameters for the metrics you want to capture, what the visualization will look like, and details like the batch sizes. To get the batches for training, call `nextTrainBatch` and reshape the Xs to the correct tensor size. You can then do exactly the same for the test data:

```
const metrics = ['loss', 'val_loss', 'accuracy', 'val_accuracy'];
const container = { name: 'Model Training', styles: { height: '640px' },
  tab: 'Training Progress' };
const fitCallbacks = tfvis.show.fitCallbacks(container, metrics);

const BATCH_SIZE = 512;
const TRAIN_DATA_SIZE = 5500;
const TEST_DATA_SIZE = 1000;

const [trainXs, trainYs] = tf.tidy(() => {
  const d = data.nextTrainBatch(TRAIN_DATA_SIZE);
  return [
    d.xs.reshape([TRAIN_DATA_SIZE, 28, 28, 1]),
    d.labels
  ];
});

const [testXs, testYs] = tf.tidy(() => {
  const d = data.nextTestBatch(TEST_DATA_SIZE);
  return [
    d.xs.reshape([TEST_DATA_SIZE, 28, 28, 1]),
    d.labels
  ];
});
```

Note the `tf.tidy` call. With TensorFlow.js this will, as its name suggests, tidy up, cleaning up all intermediate tensors except those that the function returns. It's essential when using TensorFlow.js to prevent memory leaks in the browser.

Now that you have everything set up, it's easy to do the training, giving it your training Xs and Ys (labels) as well as the validation Xs and Ys:

```
return model.fit(trainXs, trainYs, {
  batchSize: BATCH_SIZE,
  validationData: [testXs, testYs],
  epochs: 20,
  shuffle: true,
  callbacks: fitCallbacks
});
```

As you train, the callbacks will give you visualizations in the visor, as you saw back in [Figure 16-1](#).

Running Inference on Images in TensorFlow.js

To run inference, you'll first need an image. In [Figure 16-1](#), you saw an interface where an image could be drawn by hand and have inference performed on it. This uses a 280×280 canvas that is set up like this:

```
rawImage = document.getElementById('canvasimg');
ctx = canvas.getContext("2d");
ctx.fillStyle = "black";
ctx.fillRect(0,0,280,280);
```

Note that the canvas is called `rawImage`. After the user has drawn in the image (code for that is in the GitHub repo for this book), you can then run inference on it by grabbing its pixels using the `tf.browser.fromPixels` API:

```
var raw = tf.browser.fromPixels(rawImage,1);
```

It's 280×280 , so it needs to be resized to 28×28 for inference. This is done using the `tf.image.resize` APIs:

```
var resized = tf.image.resizeBilinear(raw, [28,28]);
```

The input tensor to the model is $28 \times 28 \times 1$, so you need to expand the dimensions:

```
var tensor = resized.expandDims(0);
```

Now you can predict, using `model.predict` and passing it the tensor. The output of the model is a set of probabilities, so you can pick the biggest one using TensorFlow's `argMax` function:

```
var prediction = model.predict(tensor);
var pIndex = tf.argmax(prediction, 1).dataSync();
```

The full code, including all the HTML for the page, the JavaScript for the drawing functions, as well as the TensorFlow.js model training and inference, is available in the book's [GitHub repository](#).

Summary

JavaScript is a very powerful browser-based language that can be used for many scenarios. In this chapter you took a tour of what it takes to train an image-based classifier in the browser, and then put that together with a canvas on which the user could draw. The input could then be parsed into a tensor that could be classified, with the results returned to the user. It's a useful demonstration that puts together many of the pieces of programming in JavaScript, illustrating some of the constraints that you might encounter in training, such as needing to reduce the number of HTTP connections, and how to take advantage of built-in decoders to handle data management, like you saw with the sparsely encoded labels.

You may not always want to train a new model in the browser, but instead want to reuse existing ones that you've created in TensorFlow using Python. You'll explore how to do that in the next chapter.

Reusing and Converting Python Models to JavaScript

While training in the browser is a powerful option, you may not always want to do this because of the time involved. As you saw in Chapters 15 and 16, even training simple models can lock up the browser for some time. Having a visualization of the progress helped, but it still wasn't the best of experiences. There are three alternatives to this approach. The first is to train models in Python and convert them to JavaScript. The second is to use existing models that were trained elsewhere and are provided in a JavaScript-ready format. The third is to use transfer learning, introduced in Chapter 3. In that case, features, weights, or biases that have been learned in one scenario can be transferred to another, instead of doing time-consuming relearning. We'll cover the first two cases in this chapter, and then in Chapter 18 you'll see how to do transfer learning in JavaScript.

Converting Python-Based Models to JavaScript

Models that have been trained using TensorFlow may be converted to JavaScript using the Python-based tensorflowjs tools. You can install these using:

```
!pip install tensorflowjs
```

For example, consider the following simple model that we've been using throughout the book:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

l0 = Dense(units=1, input_shape=[1])
```

```

model = Sequential([l0])
model.compile(optimizer='sgd', loss='mean_squared_error')

xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

model.fit(xs, ys, epochs=500, verbose=0)

print(model.predict([10.0]))
print("Here is what I learned: {}".format(l0.get_weights()))

```

The trained model can be saved as a saved model with this code:

```
tf.saved_model.save(model, '/tmp/saved_model/')
```

Once you have the saved model directory, you can then use the TensorFlow.js converter by passing it an input format—which in this case is a saved model—along with the location of the saved model directory and the desired location for the JSON model:

```

!tensorflowjs_converter \
  --input_format=keras_saved_model \
  /tmp/saved_model/ \
  /tmp/linear

```

The JSON model will be created in the specified directory (in this case, `/tmp/linear`). If you look at the contents of this directory, you'll also see a binary file, in this case called `group1-shardof1.bin` (Figure 17-1). This file contains the weights and biases that were learned by the network in an efficient binary format.

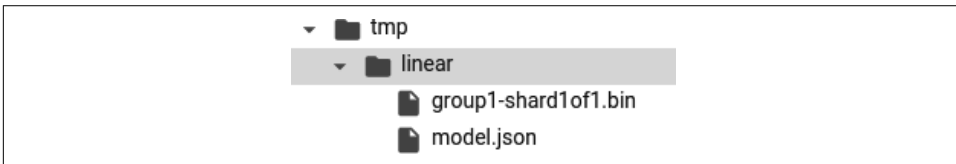


Figure 17-1. The output of the JS converter

The JSON file contains text describing the model. For example, within the JSON file you'll see a setting like this:

```

"weightsManifest": [
  {
    "paths": ["group1-shardof1.bin"],
    "weights": [{"name": "dense_2/kernel", "shape": [1, 1], "dtype": "float32"},
    {"name": "dense_2/bias", "shape": [1], "dtype": "float32"}]
  }
]

```

This indicates the location of the `.bin` file holding the weights and biases, and their shape.

If you examine the contents of the `.bin` file in a hex editor, you'll see that there are 8 bytes in it (Figure 17-2).

00000000F1 90 FF 3F 5A 4F 7D BF

Figure 17-2. The bytes in the .bin file

As our network learned with a single neuron to get $Y = 2X - 1$, the network learned a single weight as a float32 (4 bytes), and a single bias as a float32 (4 bytes). Those 8 bytes are written to the .bin file.

If you look back to the output from the code:

```
Here is what I learned: [array([[1.9966108]], dtype=float32),  
array([-0.98949206], dtype=float32)]
```

you can then convert the weight (1.9966108) to hexadecimal using a tool like the [Floating Point to Hex Converter](#) (Figure 17-3).

Floating Point to Hex Converter

☒ Show details ☐ Swap endianness

Hex value: 0xf190ff3f Convert to float

0xf190ff3f

f	1	9	0	f	f	3	f
1 1 1 1	0 0 0 1	1 0 0 1	0 0 0 0	1 1 1 1	1 1 1 1	0 0 1 1	1 1 1 1
1	11100011	001000011111111100111111					
sign	exponent	mantissa					
-1	227	1.001000011111111100111111 (binary)					
-1 *	$2^{(227 - 127)}$ *	1.1327894926071167					
-1 *	$1.26765060e+30$ *	1.1327894926071167					

1.99661

Float value: 1.99661 Convert to hex

Figure 17-3. Converting the float value to hex

You can see that the weight of 1.99661 was converted to hex of F190FF3F, which is the value of the first 4 bytes in the hex file from [Figure 17-2](#). You'll see a similar result if you convert the bias to hex (note that you'll need to swap endianness).

Using the Converted Models

Once you have the JSON file and its associated .bin file, you can use them in a TensorFlow.js app easily. To load the model from the JSON file, you specify a URL where it's hosted. If you're using the built-in server from Brackets, it will be at

127.0.0.1:<port>. When you specify this URL, you can load the model with the command `await tf.loadLayersModel(URL)`. Here's an example:

```
const MODEL_URL = 'http://127.0.0.1:35601/model.json';
const model = await tf.loadLayersModel(MODEL_URL);
```

You might need to change the 35601 to your local server port. The *model.json* file and the *.bin* file need to be in the same directory.

If you want to run a prediction using the model, you use a `tensor2d` as before, passing the input value and its shape. So, in this case, if you want to predict the value of 10.0, you can create a `tensor2d` containing `[10.0]` as the first parameter and `[1,1]` as the second:

```
const input = tf.tensor2d([10.0], [1, 1]);
const result = model.predict(input);
```

For convenience, here's the entire HTML page for this model:

```
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"></script>
<script>
  async function run(){
    const MODEL_URL = 'http://127.0.0.1:35601/model.json';
    const model = await tf.loadLayersModel(MODEL_URL);
    console.log(model.summary());
    const input = tf.tensor2d([10.0], [1, 1]);
    const result = model.predict(input);
    alert(result);
  }
  run();
</script>
<body>
</body>
</html>
```

When you run the page, it will instantly load the model and alert the results of the prediction. You can see this in [Figure 17-4](#).

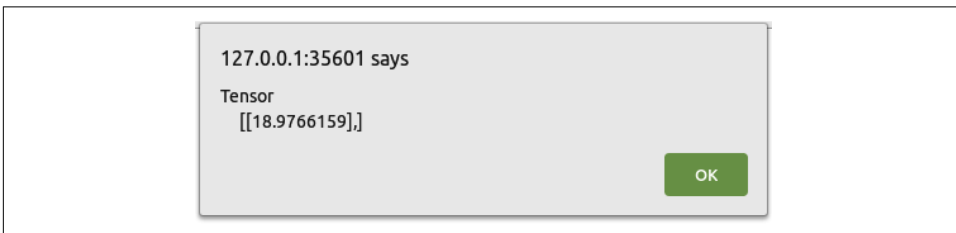


Figure 17-4. Output from the inference

Obviously this was a very simple example, where the model binary file was only 8 bytes and easy to inspect. However, hopefully it was useful to help you understand how the JSON and binary representations go hand-in-hand. As you convert your own models, you'll see much larger binary files—which ultimately are just the binary encoded weights and biases from your model, as you saw here.

In the next section you'll look at some models that have already been converted for you using this method, and how you can use them in JavaScript.

Using Preconverted JavaScript Models

In addition to being able to convert your models to JavaScript, you can also use preconverted models. The TensorFlow team has created several of these for you to try, which you can find on [GitHub](#). There are models available for different data types, including images, audio, and text. Let's explore some of these models and how you can use them in JavaScript.

Using the Toxicity Text Classifier

One of the text-based models provided by the TensorFlow team is the **Toxicity classifier**. This takes in a text string and predicts whether it contains one of the following types of toxicity:

- Identity attack
- Insult
- Obscenity
- Severe toxicity
- Sexually explicit
- Threat
- General toxicity

It's trained on the **Civil Comments dataset**, containing over two million comments that have been labeled according to these types. Using it is straightforward. You can load the model alongside TensorFlow.js like this:

```
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"></script>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow-models/toxicity"></script>
```

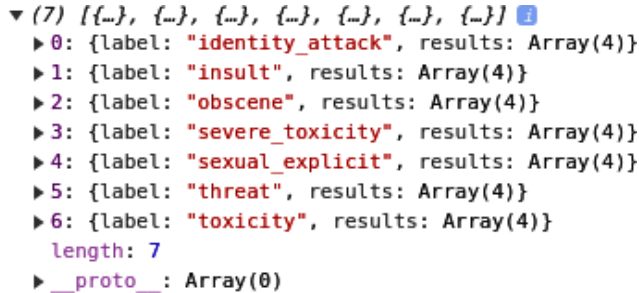
Once you have the libraries, you can set a threshold value above which the sentences will be classified. It defaults to 0.85, but you can change it to something else like this, specifying it when you load the model:

```
const threshold = 0.7;
toxicity.load(threshold).then(model => {
```

Then, if you want to classify a sentence, you can put it in an array. Multiple sentences can be classified simultaneously:

```
const sentences = ['you suck', 'I think you are stupid',  
                  'i am going to kick your head in',  
                  'you feeling lucky, punk?'];  
  
model.classify(sentences).then(predictions => {
```

At this point you can parse the predictions object for the results. It will be an array with seven entries, one for each of the toxicity types (Figure 17-5).



```
▼ (7) [{...}, {...}, {...}, {...}, {...}, {...}, {...}] ⓘ  
  ► 0: {label: "identity_attack", results: Array(4)}  
  ► 1: {label: "insult", results: Array(4)}  
  ► 2: {label: "obscene", results: Array(4)}  
  ► 3: {label: "severe_toxicity", results: Array(4)}  
  ► 4: {label: "sexual_explicit", results: Array(4)}  
  ► 5: {label: "threat", results: Array(4)}  
  ► 6: {label: "toxicity", results: Array(4)}  
    length: 7  
  ► __proto__: Array(0)
```

Figure 17-5. The results of a toxicity prediction

Within each of these are the results for each sentence against that class. So, for example, if you look at item 1, for insults, and expand it to explore the results, you'll see that there are four elements. These are the probabilities for that type of toxicity for each of the four input sentences (Figure 17-6).

The probabilities are measured as [negative, positive], so a high value in the second element indicates that type of toxicity is present. In this case, the sentence “you suck” was measured as having a .91875 probability of being an insult, whereas “I am going to kick your head in,” while toxic, shows a low level of insult probability at 0.089.

To parse these out, you can loop through the predictions array, loop through each of the insult types within the results, and then loop through their results in order to find the types of toxicity identified in each sentence. You can do this by using the `match` method, which will be positive if the predicted value is above the threshold.

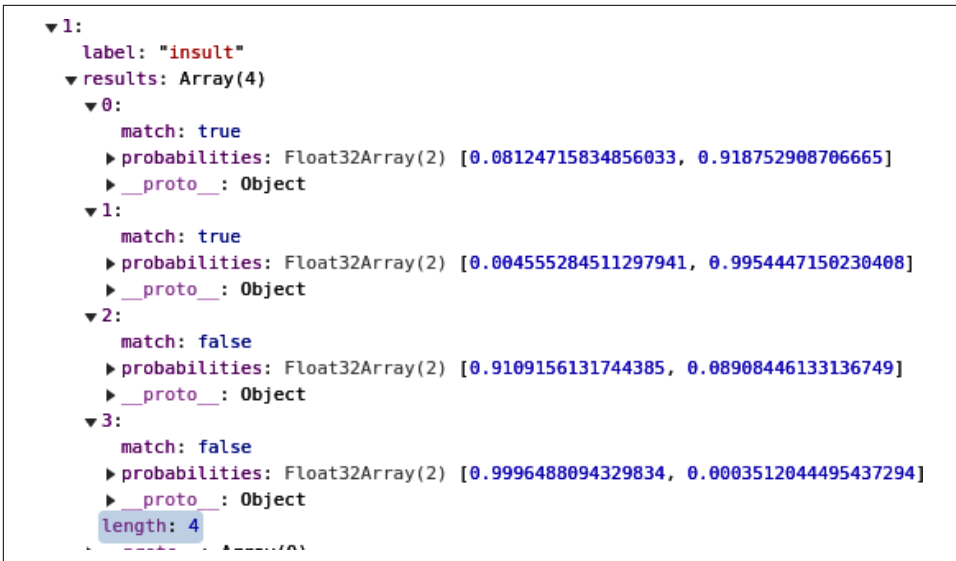


Figure 17-6. Exploring toxicity results

Here's the code:

```

for(sentence=0; sentence<sentences.length; sentence++){
  for(toxic_type=0; toxic_type<7; toxic_type++){
    if(predictions[toxic_type].results[sentence].match){
      console.log("In the sentence: " + sentences[sentence] + "\n" +
        predictions[toxic_type].label +
        " was found with probability of " +
        predictions[toxic_type].results[sentence].probabilities[1]);
    }
  }
}

```

You can see the results of this in [Figure 17-7](#).

In the sentence: you suck insult was found with probability of 0.918752908706665
In the sentence: you suck toxicity was found with probability of 0.9688231945037842
In the sentence: I think you are stupid insult was found with probability of 0.9954447150230408
In the sentence: I think you are stupid toxicity was found with probability of 0.9955390095710754
In the sentence: i am going to kick your head in toxicity was found with probability of 0.7943095564842224

Figure 17-7. Results of the Toxicity classifier on the sample input

So, if you wanted to implement some kind of toxicity filter on your site, you could do so with very few lines of code like this!

One other useful shortcut is that if you don't want to catch all seven forms of toxicity, you can specify a subset like this:

```
const labelsToInclude = ['identity_attack', 'insult', 'threat'];
```

and then specify this list alongside the threshold when loading the model:

```
toxicity.load(threshold, labelsToInclude).then(model => {})
```

The model, of course, can also be used with a Node.js backend if you want to capture and filter toxicity on the backend.

Using MobileNet for Image Classification in the Browser

As well as text classification libraries, the repository includes some libraries for image classification, such as **MobileNet**. MobileNet models are designed to be small and power-friendly while also accurate at classifying one thousand classes of image. As a result, they have one thousand output neurons, each of which is a probability that the image contains that class. So, when you pass an image to the model, you'll get back a list of one thousand probabilities that you'll need to map to these classes. However, the JavaScript library abstracts this for you, picking the top three classes in order of priority and providing just those.

Here's an excerpt from the [full list of classes](#):

```
00: background
01: tench
02: goldfish
03: great white shark
04: tiger shark
05: hammerhead
06: electric ray
```

To get started, you need to load the TensorFlow.js and mobilenet scripts, like this:

```
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"> </script>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow-models/mobilenet@1.0.0">
```

To use the model, you need to provide it with an image. The simplest way to do this is to create an `` tag and load an image into it. You can also create a `<div>` tag to hold the output:

```
<body>
  </img>
  <div id="output" style="font-family:courier;font-size:24px;height=300px">
  </div>
</body>
```

To use the model to classify the image, you then simply load it and pass a reference to the `` tag to the classifier:

```
const img = document.getElementById('img');
const outp = document.getElementById('output');
mobilenet.load().then(model => {
  model.classify(img).then(predictions => {
    console.log(predictions);
  });
});
```

This prints the output to the console log, which will look like [Figure 17-8](#).

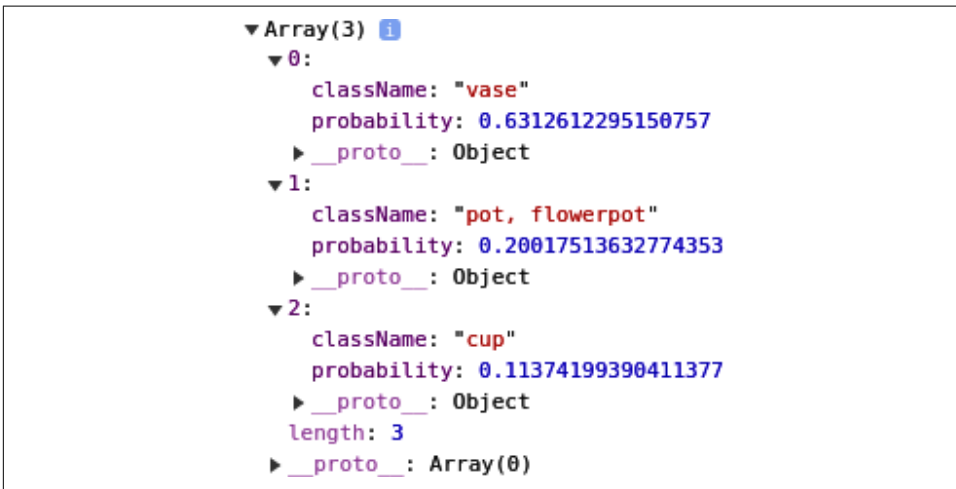


Figure 17-8. Exploring the MobileNet output

The output, as a prediction object, can also be parsed, so you can iterate through it and pick out the class name and probability like this:

```
for(var i = 0; i < predictions.length; i++){
  outp.innerHTML += "<br/>" + predictions[i].className + " : "
  + predictions[i].probability;
}
```

[Figure 17-9](#) shows the sample image in the browser alongside the predictions.



```
vase : 0.6312612295150757
pot, flowerpot : 0.20017513632774353
cup : 0.11374199390411377
```

Figure 17-9. Classifying an image

For convenience, here's the entire code listing for this simple page. To use it you need to have an image in the same directory. I'm using *coffee.jpg*, but you can of course replace the image and change the `src` attribute of the `` tag to classify something else:

```
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"> </script>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow-models/mobilenet@1.0.0">
</script>
</head>
<body>
  </img>
  <div id="output" style="font-family:courier;font-size:24px;height:300px">
    </div>
</body>
<script>
  const img = document.getElementById('img');
  const outp = document.getElementById('output');
  mobilenet.load().then(model => {
    model.classify(img).then(predictions => {
      console.log(predictions);
      for(var i = 0; i<predictions.length; i++){
        outp.innerHTML += "<br/>" + predictions[i].className + " : "
          + predictions[i].probability;
      }
    });
  });
</script>
</html>
```

Using PoseNet

Another interesting library that has been preconverted for you by the TensorFlow team is **PoseNet**, which can give you near-real-time pose estimation in the browser. It takes an image and returns a set of points for 17 body landmarks in the image:

- Nose
- Left and right eye
- Left and right ear
- Left and right shoulder
- Left and right elbow
- Left and right wrist
- Left and right hip
- Left and right knee
- Left and right ankle

For a simple scenario, we can look at estimating the pose for a single image in the browser. To do this, first load TensorFlow.js and the posenet model:

```
<head>
  <script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs"></script>
  <script src="https://cdn.jsdelivr.net/npm/@tensorflow-models/posenet">
    </script>
</head>
```

In the browser, you can load the image into an `` tag and create a canvas on which you can draw the locations of the body landmarks:

```
<div><canvas id='cnv' width='661px' height='656px' /></div>
<div></div>
```

To get the predictions, you can then just get the image element containing the picture and pass it to posenet, calling the `estimateSinglePose` method:

```
var imageElement = document.getElementById('master');
posenet.load().then(function(net) {
  const pose = net.estimateSinglePose(imageElement, {});
  return pose;
}).then(function(pose){
  console.log(pose);
  drawPredictions(pose);
})
```

This will return the predictions in an object called `pose`. This is an array of keypoints of the body parts (**Figure 17-10**).

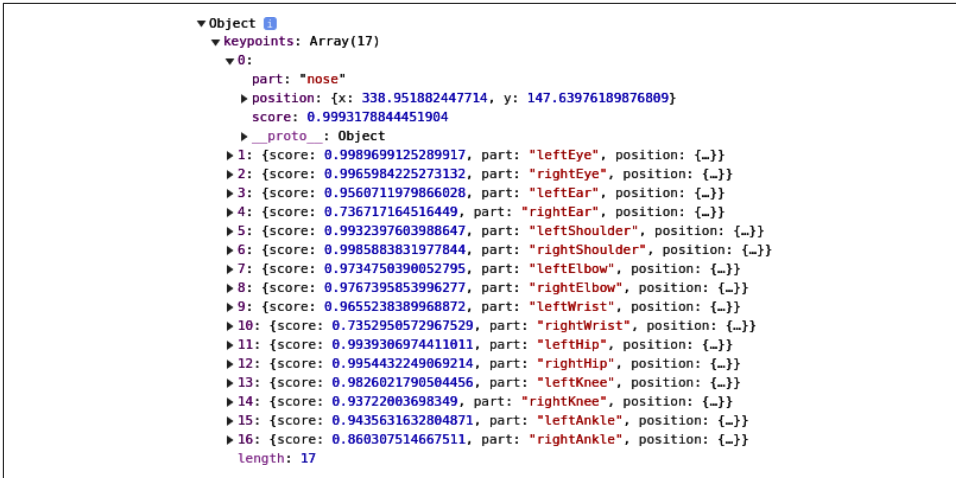


Figure 17-10. The returned positions for the pose

Each item contains a text description of the part (e.g., nose), an object with the x and y coordinates of the position, and a score value indicating the confidence that the landmark was correctly spotted. So, for example, in Figure 17-10, the likelihood that the landmark nose was spotted is .999.

You can then use these to plot the landmarks on the image. First load the image into the canvas, and then you can draw on it:

```

var canvas = document.getElementById('cnv');
var context = canvas.getContext('2d');
var img = new Image()
img.src="tennis.png"
img.onload = function(){
  context.drawImage(img, 0, 0)
  var centerX = canvas.width / 2;
  var centerY = canvas.height / 2;
  var radius = 2;

```

You can then loop through the predictions, retrieving the part names and (x,y) coordinates, and plot them on the canvas with this code:

```

for(i=0; i<pose.keypoints.length; i++){
  part = pose.keypoints[i].part
  loc = pose.keypoints[i].position;
  context.beginPath();
  context.font = "16px Arial";
  context.fillStyle="aqua"
  context.fillText(part, loc.x, loc.y)
  context.arc(loc.x, loc.y, radius, 0, 2 * Math.PI, false);
  context.fill();
}

```

Then, at runtime, you should see something like **Figure 17-11**.



Figure 17-11. Estimating and drawing body part positions on an image

You can also use the `score` property to filter out bad predictions. For example, if your picture only contains a person's face, you can update the code to filter out the low-probability predictions in order to focus on just the relevant landmarks:

```
for(i=0; i<pose.keypoints.length; i++){  
  if(pose.keypoints[i].score>0.1){  
    // Plot the points  
  }  
}
```

If the image is a closeup of somebody's face, you don't want to plot shoulders, ankles, etc. These will have very low but nonzero scores, so if you don't filter them out, they'll be plotted somewhere on the image—and as the image doesn't contain these landmarks, this will clearly be an error!

Figure 17-12 shows a picture of a face with low-probability landmarks filtered out. Note that there is no mouth landmark, because PoseNet is primarily intended for estimating body poses, not faces.

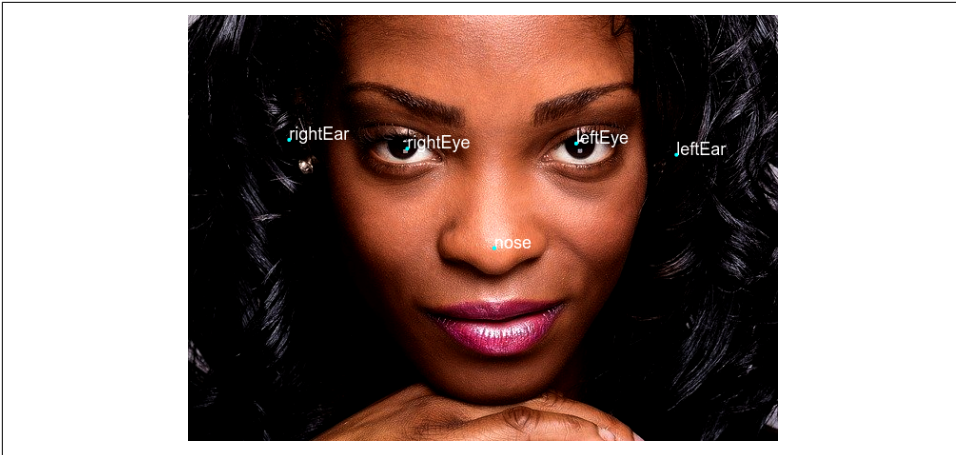


Figure 17-12. Using PoseNet on a face

There are many more features available in the PoseNet model—we’ve barely scratched the surface of what’s possible. You can do real-time pose detection in the browser with the webcam, edit the accuracy of the poses (lower-accuracy predictions can be faster), choose the architecture to optimize for speed, detect poses for multiple bodies, and much more.

Summary

In this chapter you saw how you can use TensorFlow.js with Python-created models, either by training your own model and converting it with the provided tools, or by using preexisting models. When converting, you saw how the `tensorflowjs` tools created a JSON file with the metadata of your model, as well as a binary file containing the weights and biases. It was easy to import your model as a JavaScript library and use it directly in the browser.

You then looked at a few examples of existing models that have been converted for you, and how you can incorporate them into your JavaScript code. You first experimented with the Toxicity model for processing text to identify and filter out toxic comments. You then explored using the MobileNet model for computer vision to predict the contents of an image. Finally, you saw how to use the PoseNet model to detect body landmarks in images and plot them, including how to filter out low-probability scores to avoid plotting unseen landmarks.

In [Chapter 18](#), you’ll see another method for reusing existing models: transfer learning, where you take existing pretrained features and use them in your own apps.

Transfer Learning in JavaScript

In [Chapter 17](#) you explored two methods for getting models into JavaScript: converting a Python-based model and using a preexisting model provided by the TensorFlow team. Aside from training from scratch, there's one more option: transfer learning, where a model that was previously trained for one scenario has some of its layers reused for another. For example, a convolutional neural network for computer vision might have learned several layers of filters. If it was trained on a large dataset to recognize many classes, it may have very general filters that can be used for other scenarios.

To do transfer learning with TensorFlow.js, there are a number of options, depending on how the preexisting model is distributed. The possibilities fall into three main categories:

- If the model has a *model.json* file, created by using the TensorFlow.js converter to convert it into a layers-based model, you can explore the layers, choose one of them, and have that become the input to a new model that you train.
- If the model has been converted to a graph-based model, like those commonly found on TensorFlow Hub, you can connect feature vectors from it to another model to take advantage of its learned features.
- If the model has been wrapped in a JavaScript file for easy distribution, this file will give you some handy shortcuts for prediction or transfer learning by accessing embeddings or other feature vectors.

In this chapter, you'll explore all three. We'll start by examining how you can access the prelearned layers in MobileNet, which you used as an image classifier in [Chapter 17](#), and add them to your own model.

Transfer Learning from MobileNet

The MobileNet architecture defines a **family of models** trained primarily for on-device image recognition. They're trained on the ImageNet dataset of over 10 million images, with 1,000 classes. With transfer learning, you can use their prelearned filters and change the bottom-dense layers to match your classes instead of the thousand that the model was originally trained for.

To build an app that uses transfer learning, there are a number of steps that you'll have to follow:

1. Download the MobileNet model and identify which layers to use.
2. Create your own model architecture with the outputs from MobileNet as its input.
3. Gather data into a dataset that can be used for training.
4. Train the model.
5. Run inference.

You'll go through all of these here by building a browser application that captures images from the webcam of a hand making Rock/Paper/Scissors gestures. The application then uses these to train a new model. The model will use the prelearned layers from MobileNet and add a new set of dense layers for your classes underneath.

Step 1. Download MobileNet and Identify the Layers to Use

The TensorFlow.js team hosts a number of preconverted models in Google Cloud Storage. You can find a **list of URLs** in the GitHub repo for this book, if you want to try them for yourself. There are several MobileNet models, including the one you'll use in this chapter (*mobilenet_v1_0.25_224/model.json*).

To explore the model, create a new HTML file and call it *mobilenet-transfer.html*. In this file, you'll load TensorFlow.js and an external file called *index.js* that you'll create in a moment:

```
<html>
  <head>
    <script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest">
    </script>
  </head>
  <body></body>
  <script src="index.js"></script>
</html>
```

Next, create the *index.js* file that the preceding HTML is referring to. This will contain an asynchronous method that downloads the model and prints its summary:

```

async function init(){
  const url = 'https://storage.googleapis.com/tfjs-
models/tfjs/mobilenet_v1_0.25_224/model.json'

  const mobilenet = await tf.loadLayersModel(url);
  console.log(mobilenet.summary())
}

init()

```

If you look at the `model.summary` output in the console and scroll to the bottom, you'll see something like [Figure 18-1](#).

			tfjs@latest:2
conv_pw_13_bn (BatchNormaliz	[null,7,7,256]	1024	tfjs@latest:2
			tfjs@latest:2
conv_pw_13_relu (Activation)	[null,7,7,256]	0	tfjs@latest:2
			tfjs@latest:2
global_average_pooling2d_1 ([null,256]	0	tfjs@latest:2
			tfjs@latest:2
reshape_1 (Reshape)	[null,1,1,256]	0	tfjs@latest:2
			tfjs@latest:2
dropout (Dropout)	[null,1,1,256]	0	tfjs@latest:2
			tfjs@latest:2
conv_preds (Conv2D)	[null,1,1,1000]		tfjs@latest:2
257000			tfjs@latest:2
			tfjs@latest:2
act_softmax (Activation)	[null,1,1,1000]	0	tfjs@latest:2
			tfjs@latest:2
reshape_2 (Reshape)	[null,1000]	0	tfjs@latest:2
			tfjs@latest:2
=====			tfjs@latest:2
Total params: 475544			tfjs@latest:2
Trainable params: 470072			tfjs@latest:2
Non-trainable params: 5472			tfjs@latest:2
			tfjs@latest:2

Figure 18-1. Output of `model.summary` for the MobileNet JSON model

The key to transfer learning with MobileNet is to look for the *activation* layers. As you can see, there are two right at the bottom. The last one has one thousand outputs, which corresponds to the thousand classes that MobileNet supports. So, if you want the learned activation layers—in particular, the learned convolutional filters—look for activation layers above this, and note their names. As you can see in [Figure 18-1](#), the last activation layer in the model, before the final one, is called `conv_pw_13_relu`.

You can use that (or indeed, any activation layers before it) as your outputs from the model if you want to do transfer learning.

Step 2. Create Your Own Model Architecture with the Outputs from MobileNet as Its Input

When designing a model, you typically design all your layers, starting with the input layer and ending with the output one. With transfer learning, you will pass input to the model from which you are transferring, and you will create new output layers. Consider [Figure 18-2](#)—this is the rough, high-level architecture of MobileNet. It takes in images of dimension $224 \times 224 \times 3$ and passes them through a neural network architecture, giving you an output of one thousand values, each of which is the probability that the image contains the relevant class.

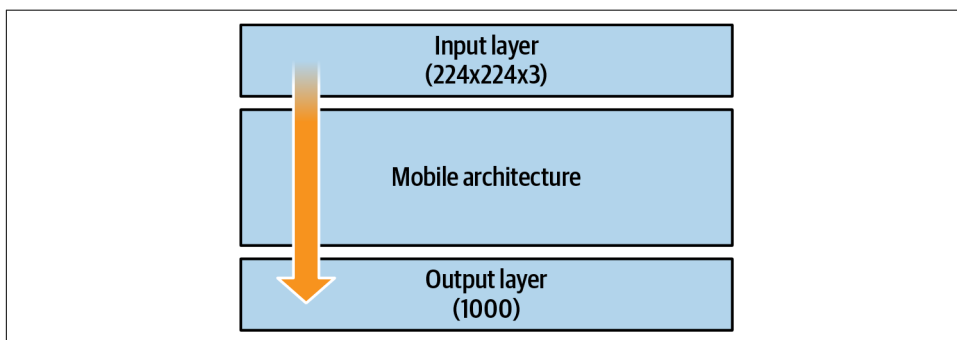


Figure 18-2. High-level MobileNet architecture

Earlier you looked at the innards of that architecture and identified the last activation convolutional layer, called `conv_pw_13_relu`. You can see what the architecture looks like with this layer included in [Figure 18-3](#).

The MobileNet architecture still has one thousand classes that it recognizes, none of which are the ones you want to implement (a hand making gestures for the game Rock Paper Scissors). You'll need a new model that is trained on these three classes. You could train it from scratch and have it learn all the filters that will give you the features to distinguish them, as seen in earlier chapters. Or you can take the pre-learned filters from MobileNet, using the architecture all the way up to `conv_pw_13_relu`, and feed that to a new model that classifies only three classes. See [Figure 18-4](#) for an abstraction of this.

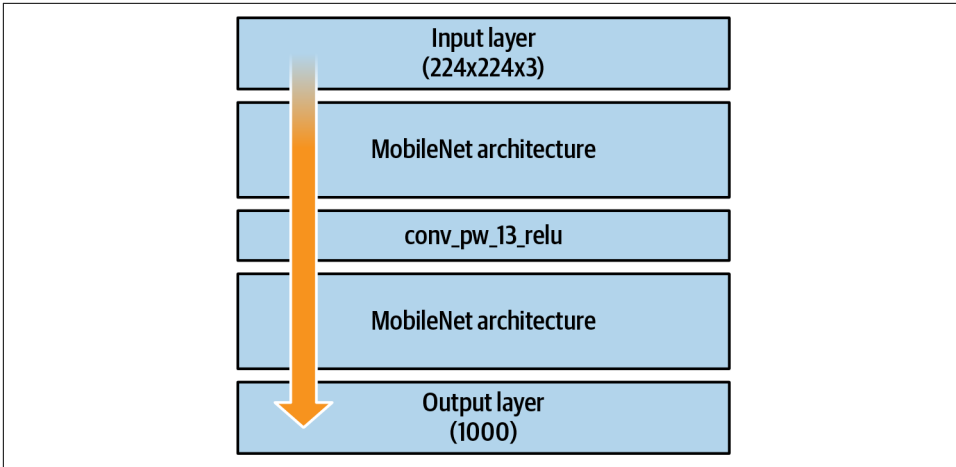


Figure 18-3. High-level MobileNet architecture showing `conv_pw_13_relu`

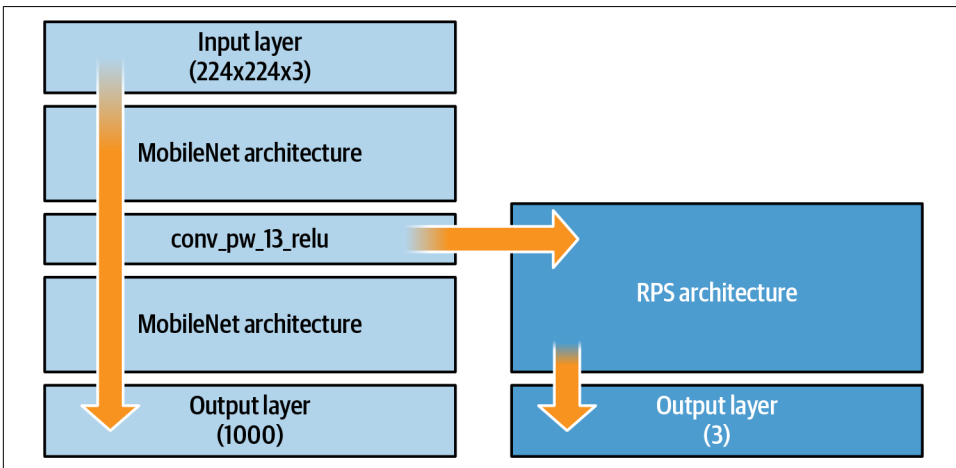


Figure 18-4. Transfer learning from `conv_pw_13_relu` to a new architecture

To implement this in code, you can update your `index.js` to the following:

```
let mobilenet

async function loadMobilenet() {
  const mobilenet = await tf.loadLayersModel(url);
  const layer = mobilenet.getLayer('conv_pw_13_relu');
  return tf.model({inputs: mobilenet.inputs, outputs: layer.output});
}

async function init(){
  mobilenet = await loadMobilenet()
  model = tf.sequential({
```

```

layers: [
  tf.layers.flatten({inputShape: mobilenet.outputs[0].shape.slice(1)}),
  tf.layers.dense({ units: 100, activation: 'relu'}),
  tf.layers.dense({ units: 3, activation: 'softmax'})
]
});
console.log(model.summary())
}

init()

```

The loading of mobilenet has been put into its own async function. Once the model has finished loading, the `conv_pw_13_relu` layer can be extracted from it using the `getLayer` method. The function will then return a model with its inputs set to mobilenet's inputs and its outputs set to `conv_pw_13_relu`'s outputs. This is visualized by the right-pointing arrow in [Figure 18-4](#).

Once this function returns, you can create a new sequential model. Note the first layer in it—it's a flatten of the mobilenet outputs (i.e., the `conv_pw_13_relu` outputs), which then feeds into a dense layer of one hundred neurons, which feeds into a dense layer of three neurons (one each for Rock, Paper, and Scissors).

If you now do a `model.fit` on this model, you'll train it to recognize three classes—but instead of learning all the filters to identify features in the image from scratch, you'll be able to use the ones that were previously learned by MobileNet. Before you can do that, though, you'll need some data. You'll see how to gather that in the next step.

Step 3. Gather and Format the Data

For this example, you'll use the webcam in the browser to capture images of a hand making Rock/Paper/Scissors gestures. Capturing data from the webcam is beyond the scope of this book, so I won't go into detail on it here, but there's a `webcam.js` file in the GitHub repo for this book (created by the TensorFlow team) that can handle everything for you. This captures images from the webcam and returns them in a TensorFlow-friendly format as batched images. It also handles all the tidying-up code from `TensorFlow.js` that the browser will need to avoid memory leaks. Here's a snippet from the file:

```

capture() {
  return tf.tidy(() => {
    // Read the image as a tensor from the webcam <video> element.
    const webcamImage = tf.browser.fromPixels(this.webcamElement);
    const reversedImage = webcamImage.reverse(1);
    // Crop the image so we're using the center square of the rectangle.
    const croppedImage = this.cropImage(reversedImage);
    // Expand the outermost dimension so we have a batch size of 1.
    const batchedImage = croppedImage.expandDims(0);

```

```

    // Normalize the image between -1 and 1. The image comes in between
    // 0-255, so we divide by 127 and subtract 1.
    return batchedImage.toFloat().div(tf.scalar(127)).sub(tf.scalar(1));
  });
}

```

You can include this `.js` file in your HTML with a simple `<script>` tag:

```
<script src="webcam.js"></script>
```

You can then update the HTML with a `<div>` to hold the video preview from the webcam, buttons that the user will select to capture samples of Rock/Paper/Scissors gestures, and `<div>`s to output the number of samples captured. Here's what it should look like:

```

<html>
  <head>
    <script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest">
    </script>
    <script src="webcam.js"></script>
  </head>
  <body>
    <div>
      <video autoplay playsinline muted id="wc" width="224" height="224"/>
    </div>
    <button type="button" id="0" onclick="handleButton(this)">Rock</button>
    <button type="button" id="1" onclick="handleButton(this)">Paper</button>
    <button type="button" id="2" onclick="handleButton(this)">Scissors</button>
    <div id="rocksamples">Rock Samples:</div>
    <div id="papersamples">Paper Samples:</div>
    <div id="scissorssamples">Scissors Samples:</div>
  </body>
  <script src="index.js"></script>
</html>

```

Then all you need to do is add a `const` to the top of your `index.js` file that initializes the webcam with the ID of the `<video>` tag in your HTML:

```
const webcam = new Webcam(document.getElementById('wc'));
```

You can then initialize the webcam within your `init` function:

```
await webcam.setup();
```

Running the page will now give you a webcam preview along with the three buttons (see [Figure 18-5](#)).

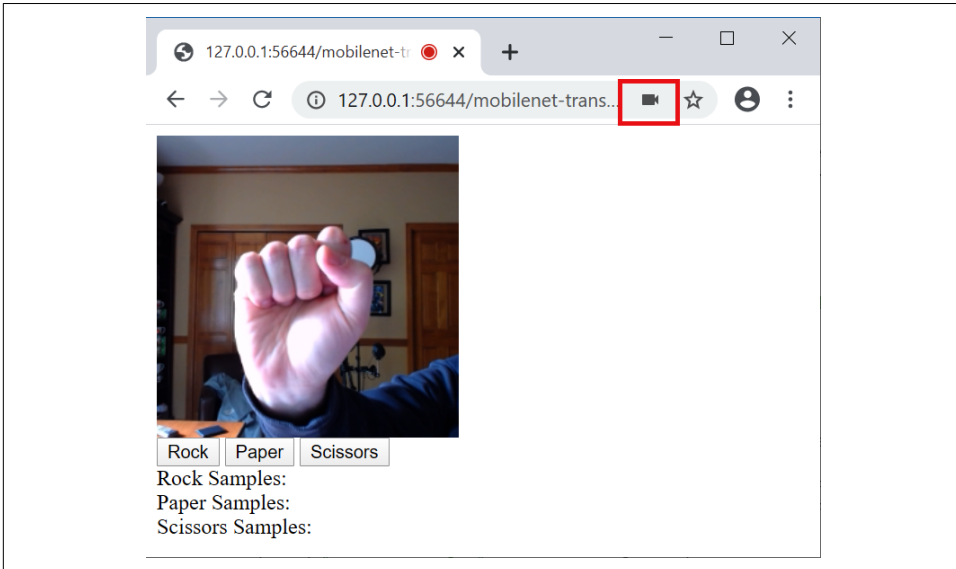


Figure 18-5. Getting the webcam preview to work

Note that if you don't see the preview, you should see an icon like the one highlighted at the top of this figure in the Chrome status bar. If it has a red line through it, you need to give the browser permission to use the webcam, after which you should see the preview. The next thing you need to do is capture the images and put them in a format that makes it easy for you to train the model you created in step 2.

As TensorFlow.js cannot take advantage of built-in datasets like Python, you'll have to roll your own dataset class. Fortunately, it's not as difficult as it sounds. In JavaScript, create a new file called *rps-dataset.js*. Construct it with an array for the labels, as follows:

```
class RPSDataset {  
  constructor() {  
    this.labels = []  
  }  
}
```

Every time you capture a new example of a Rock/Paper/Scissors gesture from the webcam, you'll want to add it to the dataset. This can be achieved with an `addExample` method. The examples will be added as `xs`. Note that this will not add the raw image, but the classification of the image by the truncated `mobilenet`. You'll see this in a moment.

The first time you call this, the `xs` will be `null`, so you'll create the `xs` using the `tf.keep` method. This, as its name suggests, prevents a tensor from being destroyed in a `tf.tidy` call. It also pushes the label to the `labels` array created in the constructor. For subsequent calls, the `xs` will not be `null`, so you'll copy the `xs` into an `oldX` and then `concat` the example to that, making that the new `xs`. You'll then push the label to the `labels` array and dispose of the old `xs`:

```
addExample(example, label) {
  if (this.xs == null) {
    this.xs = tf.keep(example);
    this.labels.push(label);
  } else {
    const oldX = this.xs;
    this.xs = tf.keep(oldX.concat(example, 0));
    this.labels.push(label);
    oldX.dispose();
  }
}
```

Following this approach, your labels will be an array of values. But to train the model you need them as a one-hot encoded array, so you'll need to add a helper function to your dataset class. This JavaScript will encode the `labels` array into a number of classes specified by the `numClasses` parameter:

```
encodeLabels(numClasses) {
  for (var i = 0; i < this.labels.length; i++) {
    if (this.ys == null) {
      this.ys = tf.keep(tf.tidy(
        () => {return tf.oneHot(
          tf.tensor1d([this.labels[i]].toInt(), numClasses)}));
    } else {
      const y = tf.tidy(
        () => {return tf.oneHot(
          tf.tensor1d([this.labels[i]].toInt(), numClasses)});
      const oldY = this.ys;
      this.ys = tf.keep(oldY.concat(y, 0));
      oldY.dispose();
      y.dispose();
    }
  }
}
```

The key is the `tf.oneHot` method, which, as its name suggests, encodes its given parameters into a one-hot encoding.

In your HTML you added the three buttons and specified their `onclick` to call a function called `handleButton`, like this:

```
<button type="button" id="0" onclick="handleButton(this)">Rock</button>
<button type="button" id="1" onclick="handleButton(this)">Paper</button>
<button type="button" id="2" onclick="handleButton(this)">Scissors</button>
```


You can implement this in your *index.js* script by switching on the element ID (which is 0, 1, or 2 for Rock, Paper, and Scissors, respectively), turning that into a label, capturing the webcam image, calling the `predict` method on `mobilenet`, and adding the result as an example to the dataset using the method you created earlier:

```
function handleButton(elem){  
  label = parseInt(elem.id);  
  const img = webcam.capture();  
  dataset.addExample(mobilenet.predict(img), label);  
}
```

It's worth making sure you understand the `addExample` method before going further. While you *could* create a dataset that captures the raw images and adds them to a dataset, recall [Figure 18-4](#). You created the `mobilenet` object with the output of `conv_pw_13_relu`. By calling `predict`, you'll get the output of that layer. And if you look back to [Figure 18-1](#), you'll see that the output was `[?, 7, 7, 256]`. This is summarized in [Figure 18-6](#).

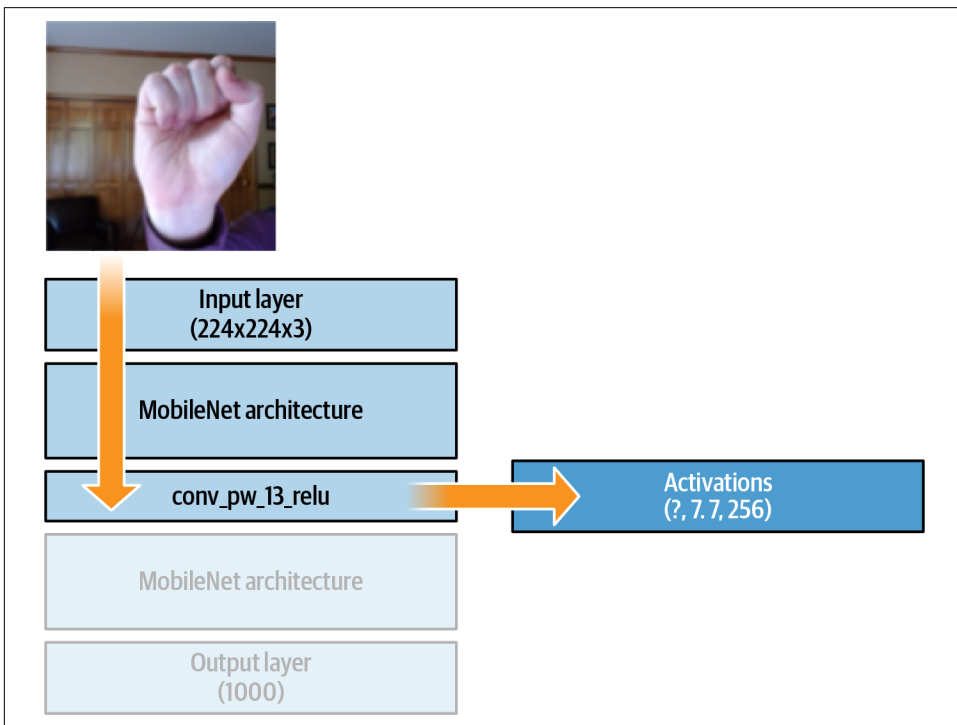


Figure 18-6. Results of `mobilenet.predict`

Recall that with a CNN, as the image progresses through the network a number of filters are learned, and the results of these filters are multiplied into the image. They usually get pooled and passed to the next layer. With this architecture, by the time the

image reaches the output layer, instead of a single color image you'll have $256 \times 7 \times 7$ images, which are the results of all the filter applications. These can then be fed into a dense network for classification.

You can also add code to this to update the user interface, counting the number of samples added. I've omitted it here for brevity, but it's all in the GitHub repo.

Don't forget to add the `rps-dataset.js` file to your HTML using a `<script>` tag:

```
<script src="rps-dataset.js"></script>
```

In the Chrome developer tools, you can add breakpoints and watch variables. Run your code, and add a watch to the `dataset` variable, and a breakpoint to the `dataset.addExample` method. Click one of the Rock/Paper/Scissors buttons and you'll see the dataset get updated. In Figure 18-7 you can see the results after I clicked each of the three buttons.

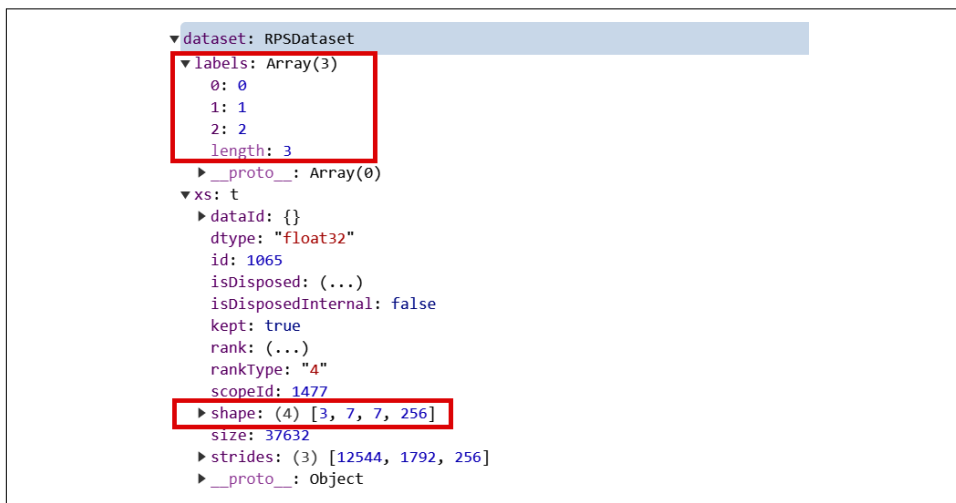


Figure 18-7. Exploring the dataset

Note that the `labels` array is set up with 0, 1, 2 for the three labels. It hasn't been one-hot encoded yet. Also, within the dataset you can see a 4D tensor with all of the gathered data. The first dimension (3) is the number of samples gathered. The subsequent dimensions (7, 7, 256) are the activations from `mobilenet`.

You now have a dataset that can be used to train your model. At runtime you can have your users click each of the buttons to gather a number of samples of each type, which will then be fed into the dense layers that you specified for classification.

Step 4. Train the Model

This app will work by having a button to train the model. Once the training is complete, you can press a button to start the model predicting what it sees in the webcam, and another to stop predicting.

Add the following HTML to your page to add these three buttons, and some `<div>` tags to hold the output. Note that the buttons call methods called `doTraining`, `startPredicting`, and `stopPredicting`:

```
<button type="button" id="train" onclick="doTraining()">
  Train Network
</button>
<div id="dummy">
  Once training is complete, click 'Start Predicting' to see predictions
  and 'Stop Predicting' to end
</div>
<button type="button" id="startPredicting" onclick="startPredicting()">
  Start Predicting
</button>
<button type="button" id="stopPredicting" onclick="stopPredicting()">
  Stop Predicting
</button>
<div id="prediction"></div>
```

Within your `index.js`, you can then add a `doTraining` method and populate it:

```
function doTraining(){
  train();
}
```

Within the `train` method you can then define your model architecture, one-hot encode the labels, and train the model. Note the first layer in the model has its input Shape defined as the output shape of `mobilenet`, and you previously specified the `mobilenet` object's output to be the `conv_pw_13_relu`:

```
async function train() {
  dataset.js = null;
  dataset.encodeLabels(3);
  model = tf.sequential({
    layers: [
      tf.layers.flatten({inputShape: mobilenet.outputs[0].shape.slice(1)}),
      tf.layers.dense({ units: 100, activation: 'relu'}),
      tf.layers.dense({ units: 3, activation: 'softmax'})
    ]
  });
  const optimizer = tf.train.adam(0.0001);
  model.compile({optimizer: optimizer, loss: 'categoricalCrossentropy'});
  let loss = 0;
  model.fit(dataset.xs, dataset.js, {
    epochs: 10,
    callbacks: {
```

```

        onBatchEnd: async (batch, logs) => {
            loss = logs.loss.toFixed(5);
            console.log('LOSS: ' + loss);
        }
    }
});
}

```

This will train the model for 10 epochs. You can adjust this as you see fit depending on the loss in the model.

Earlier you defined the model in *init.js*, but it's a good idea to move it here instead and keep the *init* function just for initialization. So, your *init* should look like this:

```

async function init(){
    await webcam.setup();
    mobilenet = await loadMobilenet()
}

```

At this point you can practice making Rock/Paper/Scissors gestures in front of the webcam. Press the appropriate button to capture an example of a given class. Repeat this about 50 times for each of the classes, and then press Train Network. After a few moments, the training will complete, and in the console you'll see the loss values. In my case the loss started at about 2.5 and ended at 0.0004, indicating that the model was learning well.

Note that 50 samples is more than enough for each class, because when we add the examples to the dataset, we add the *activated* examples. Each image gives us $256 \times 7 \times 7$ images to feed into the dense layers, so 150 samples gives us 38,400 overall items for training.

Now that you have a trained model, you can try doing predictions with it!

Step 5. Run Inference with the Model

Having completed step 4, you should have code that gives you a fully trained model. You also created HTML buttons to start and stop predicting. These were configured to call the *startPredicting* and *stopPredicting* methods, so you should create them now. Each one should just set an *isPredicting* Boolean to true/false, respectively, for whether you want to predict or not. After that they call the *predict* method:

```

function startPredicting(){
    isPredicting = true;
    predict();
}

function stopPredicting(){
    isPredicting = false;
}

```

```

    predict();
}

```

The `predict` method then can use your trained model. It will capture the webcam input and get the activations by calling `mobilenet.predict` with the image. Then, once it has the activations, it can pass them to the model to get a prediction. As the labels were one-hot encoded, you can call `argMax` on the predictions to get the likely output:

```

async function predict() {
  while (isPredicting) {
    const predictedClass = tf.tidy(() => {
      const img = webcam.capture();
      const activation = mobilenet.predict(img);
      const predictions = model.predict(activation);
      return predictions.as1D().argMax();
    });
    const classId = (await predictedClass.data())[0];
    var predictionText = "";
    switch(classId){
      case 0:
        predictionText = "I see Rock";
        break;
      case 1:
        predictionText = "I see Paper";
        break;
      case 2:
        predictionText = "I see Scissors";
        break;
    }
    document.getElementById("prediction").innerText = predictionText;

    predictedClass.dispose();
    await tf.nextFrame();
  }
}

```

With 0, 1, or 2 as the result, you can then write the value to the prediction `<div>` and clean up.

Note that this is gated on the `isPredicting` Boolean, so you can turn predictions on or off with the relevant buttons. Now when you run the page you can collect samples, train the model, and run inference. See [Figure 18-8](#) for an example, where it classified my hand gesture as Scissors!

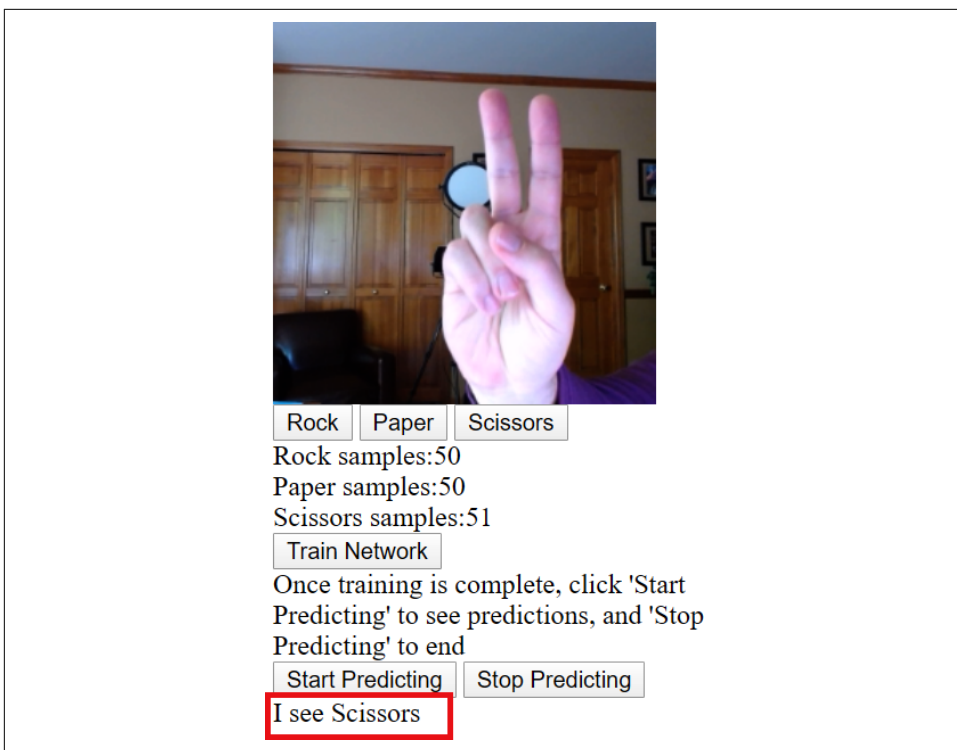


Figure 18-8. Running inference in the browser with the trained model

From this example, you saw how to build your own model for transfer learning. Next you'll explore an alternative approach using graph-based models stored in TensorFlow Hub.

Transfer Learning from TensorFlow Hub

TensorFlow Hub is an online library of reusable TensorFlow models. Many of the models have already been converted into JavaScript for you—but when it comes to transfer learning, you should look for “image feature vector” model types, and not the full models themselves. These are models that have already been pruned to output the learned features. The approach here is a little different from the example in the previous section, where you had activations output from MobileNet that you could then transfer to your custom model. Instead, a *feature vector* is a 1D tensor that represents the entire image.

To find a MobileNet model to experiment with, visit [TFHub.dev](https://tfhub.dev), choose TF.js as the model format you want, and select the MobileNet architecture. You'll see lots of options of models that are available to you, as shown in [Figure 18-9](#).

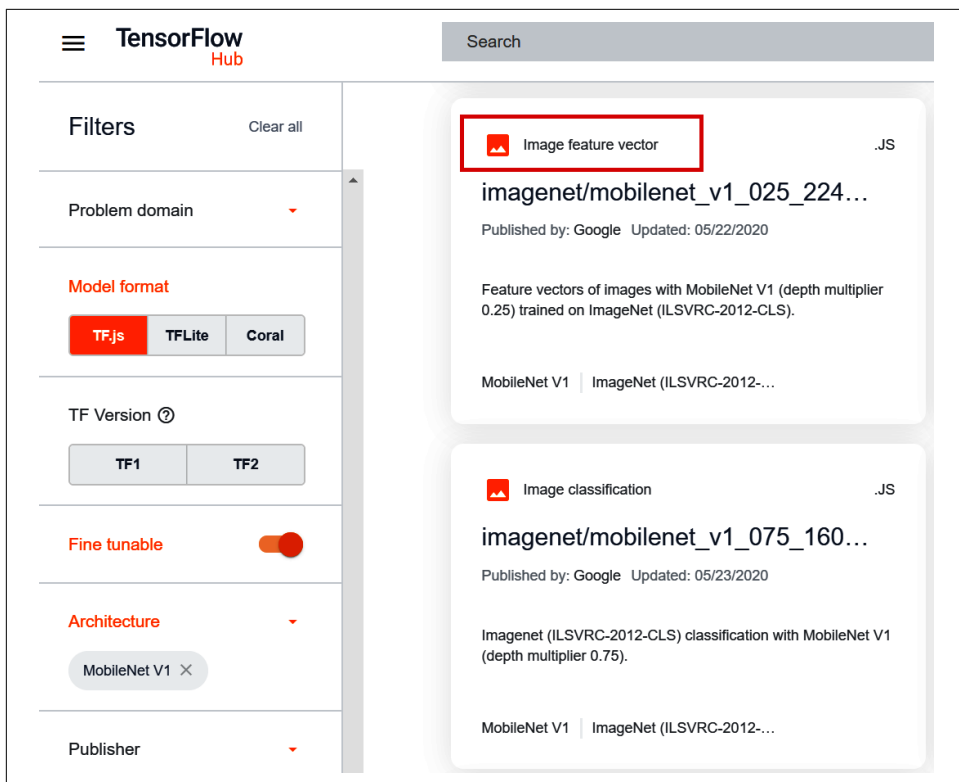


Figure 18-9. Using TFHub.dev to find JavaScript models

Find an image feature vector model (I'm using 025_224), and select it. In the “Example use” section on the model's details page, you'll find code indicating how to download the image—for example:

```
tf.loadGraphModel("https://tfhub.dev/google/tfjs-model/imagenet/
mobilenet_v1_025_224/feature_vector/3/default/1", { fromTFHub: true })
```

You can use this to download the model so you can inspect the dimensions of the feature vector. Here's a simple HTML file with this code in it that classifies an image called *dog.jpg*, which should be in the same directory:

```
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"> </script>
</head>
<body>
  
</body>
</html>
<script>
  async function run(){
```

```

const img = document.getElementById('img');
model = await tf.loadGraphModel('https://tfhub.dev/google/tfjs-model/imagenet/
mobilenet_v1_025_224/feature_vector/3/default/1', {fromTFHub: true});
var raw = tf.browser.fromPixels(img).toFloat();
var resized = tf.image.resizeBilinear(raw, [224, 224]);
var tensor = resized.expandDims(0);
var result = await model.predict(tensor).data();
console.log(result)
}

run();

</script>

```

When you run this and look in the console, you'll see the output from this classifier (Figure 18-10). If you are using the same model as me, you should see a Float32Array with 256 elements in it. Other MobileNet versions may have output of different sizes.

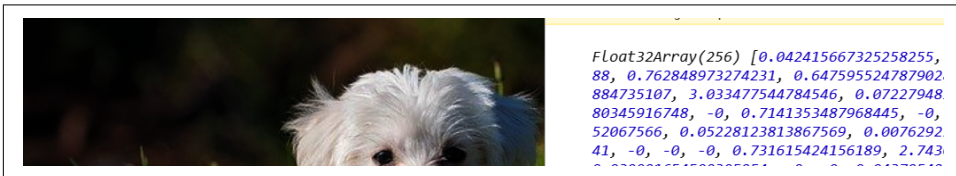


Figure 18-10. Exploring the console output

Once you know the output shape of the image feature vector model, you can use it for transfer learning. So, for example, for the Rock/Paper/Scissors example, you could use an architecture like that in Figure 18-11.

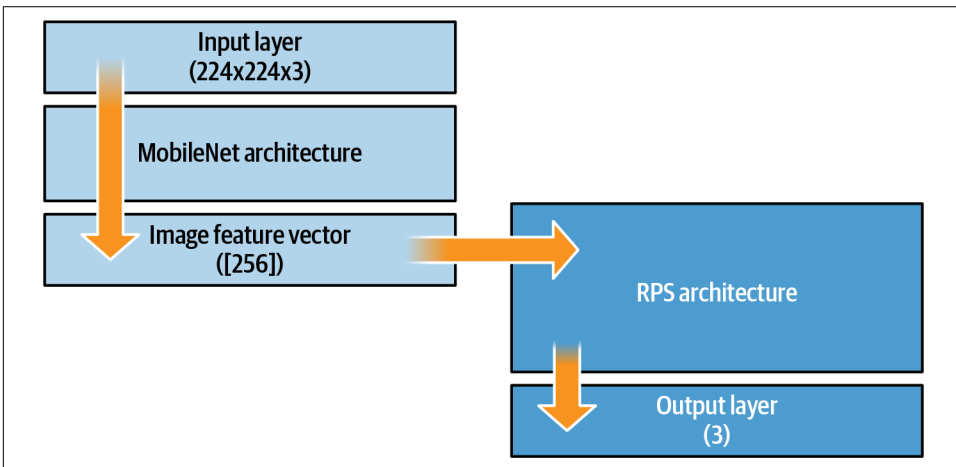


Figure 18-11. Transfer learning using image feature vectors

Now you can edit the code for your transfer learning Rock/Paper/Scissors app by changing how and where you load the model from, and by amending the classifier to accept the image feature vector instead of the activated features, as earlier.

If you want to load your model from TensorFlow Hub, just update the `loadMobileNet` function like this:

```
async function loadMobileNet() {
  const mobilenet =
    await tf.loadGraphModel("https://tfhub.dev/google/tfjs-model/imagenet/
      mobilenet_v1_050_160/feature_vector/3/default/1", {fromTFHub: true})
  return mobilenet
}
```

And then, in your `train` method, where you define the model for the classification, you update it to receive the output from the image feature vector ([256]) to the first layer. Here's the code:

```
model = tf.sequential({
  layers: [
    tf.layers.dense({ inputShape: [256], units: 100, activation: 'relu'}),
    tf.layers.dense({ units: 3, activation: 'softmax'})
  ]
});
```

Note that this shape will be different for different models. You can use code like the HTML shown earlier to find it if it isn't published for you.

Once this is done, you can do transfer learning from the model in TensorFlow Hub using JavaScript!

Using Models from TensorFlow.org

Another source for **models for JavaScript developers** is TensorFlow.org (see **Figure 18-12**). The models provided here, for image classification, object detection, and more, are ready for immediate use. Clicking any of the links will take you to a GitHub repository of JavaScript classes that wrap the graph-based models with logic to make using them much easier.

In the case of **MobileNet**, you can use the model with a `<script>` include like this:

```
<script src="https://cdn.jsdelivr.net/npm/@tensorflow-models/mobilenet@1.0.0">
</script>
```

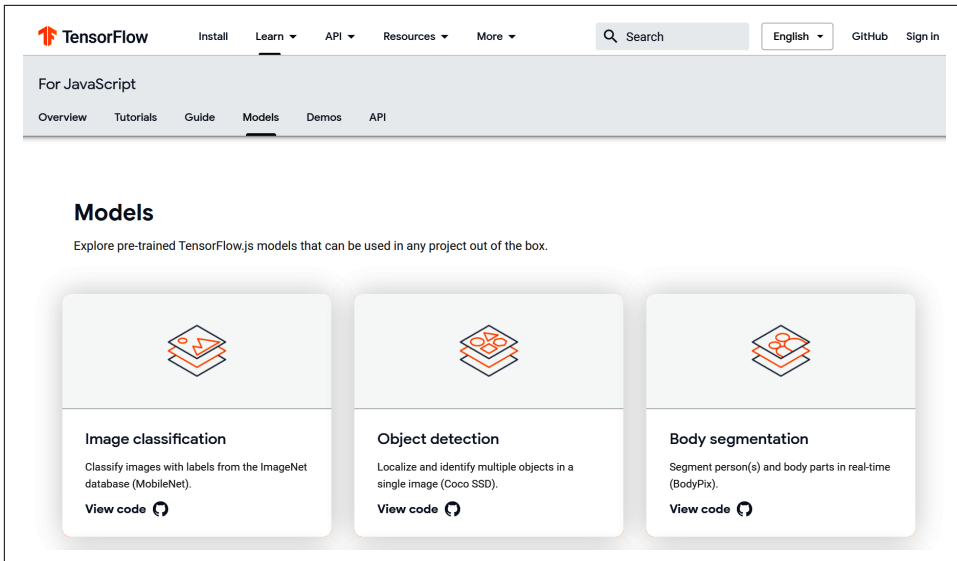


Figure 18-12. Browsing models on *TensorFlow.org*

If you take a look at the code, you'll notice two things. First, the set of labels is encoded within the JavaScript, giving you a handy way of seeing the inference results without a secondary lookup. You can see a snippet of the code here:

```
var i={
  0:"tench, Tinca tinca",
  1:"goldfish, Carassius auratus",
  2:"great white shark, white shark, man-eater, man-eating shark, Carcharodon
    carcharias",
  3:"tiger shark, Galeocerdo cuvieri"
  ...
}
```

Additionally, toward the bottom of this file you'll see a number of models, layers, and activations from TensorFlow Hub that you can load into JavaScript variables. For example, for version 1 of MobileNet you may see an entry like this:

```
>n={"1.00">:
>{>.25>:">https://tfhub.dev/google/imagenet/mobilenet\_v1\_025\_224/classification/1",
  "0.50">:">https://tfhub.dev/google/imagenet/mobilenet\_v1\_050\_224/classification/1",
  >.75>:">https://tfhub.dev/google/imagenet/mobilenet\_v1\_075\_224/classification/1",
  "1.00">:">https://tfhub.dev/google/imagenet/mobilenet\_v1\_100\_224/classification/1"
>}
```

The values 0.25, 0.50, 0.75, etc. are “width multiplier” values. These are used to construct smaller, less computationally expensive models; you can find details in the [original paper](#) introducing the architecture.

The code offers many handy shortcuts. For example, when running inference on an image, compare the following listing to the one shown a little earlier, where you used MobileNet to get an inference for the dog image. Here's the full HTML:

```
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest">
</script>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow-models/mobilenet@1.0.0">
</script>
</head>
<body>
  
</body>
</html>
<script>
async function run(){
  const img = document.getElementById('img');
  mobilenet.load().then(model => {
    model.classify(img).then(predictions => {
      console.log('Predictions: ');
      console.log(predictions);
    });
  });
}

run();

</script>
```

Note how you don't have to preconvert the image into tensors in order to do the classification. This code is much cleaner, and allows you to just focus on your predictions. To get embeddings, you can use `model.infer` instead of `model.classify` like this:

```
embeddings = model.infer(img, embedding=true);
console.log(embeddings);
```

So, if you wish, you could create a transfer learning scenario from MobileNet using these embeddings.

Summary

In this chapter you took a look at a variety of options for transfer learning from pre-existing JavaScript-based models. As there's a range of different model implementation types, there are also a number of options for accessing them for transfer learning. First, you saw how to use the JSON files created by the TensorFlow.js converter to explore the layers of the model and choose one to transfer from. The second option was to use graph-based models. This is the favored type of model on TensorFlow Hub

(because they generally give faster inference), but you lose a little of the flexibility of being able to choose the layer to do the transfer learning from. When using this method, the JavaScript bundle that you download doesn't contain the full model, but instead truncates it as the feature vector output. You transfer from this to your own model. Finally, you saw how to work with the prewrapped JavaScript models made available by the TensorFlow team on [TensorFlow.org](https://www.tensorflow.org), which include helper functions for accessing the data, inspecting the classes, as well as getting the embeddings or other feature vectors from the model so they can be used for transfer learning.

In general, I would recommend taking the TensorFlow Hub approach and using models with prebuilt feature vector outputs when they're available—but if they aren't, it's good to know that TensorFlow.js has a flexible enough ecosystem to allow for transfer learning in a variety of ways.

Deployment with TensorFlow Serving

Over the past few chapters you’ve been looking at deployment surfaces for your models—on Android and iOS, and in the web browser. Another obvious place where models can be deployed is to a server, so that your users can pass data to your server and have it run inference using your model and return the results. This can be achieved using TensorFlow Serving, a simple “wrapper” for models that provides an API surface as well as production-level scalability. In this chapter you’ll get an introduction to TensorFlow Serving and how you can use it to deploy and manage inference with a simple model.

What Is TensorFlow Serving?

This book has primarily focused on the code for creating models, and while this is a massive undertaking in itself, it’s only a small part of the overall picture of what it takes to use machine learning models in production. As you can see in [Figure 19-1](#), your code needs to work alongside code for configuration, data collection, data verification, monitoring, machine resource management, and feature extraction, as well as analysis tools, process management tools, and serving infrastructure.

TensorFlow’s ecosystem for these tools is called *TensorFlow Extended* (TFX). Other than the serving infrastructure, covered in this chapter, I won’t be going into any of the rest of TFX. A great resource if you’d like to learn more about it is the book *Building Machine Learning Pipelines* by Hannes Hapke and Catherine Nelson (O’Reilly).

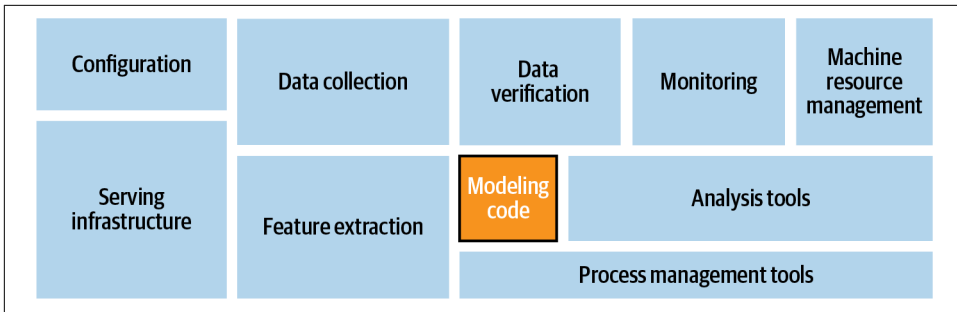


Figure 19-1. System architecture modules for ML systems

The pipeline for machine learning models is summarized in [Figure 19-2](#).

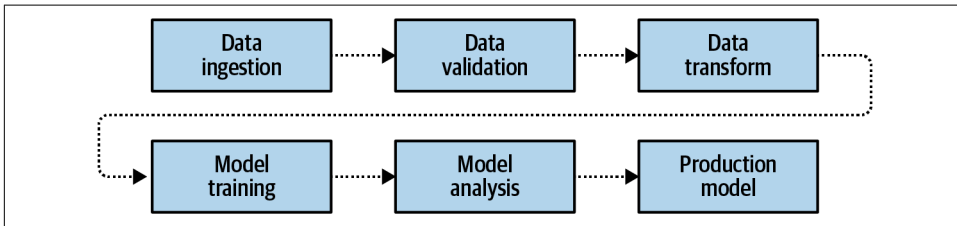


Figure 19-2. Machine learning production pipeline

The pipeline calls for data to first be gathered and ingested, and then validated. Once it's “clean,” the data is transformed into a format that can be used for training, including labeling it as appropriate. From there, models can be trained, and upon completion they'll be analyzed. You've been doing that already when testing your models for accuracy, looking at the loss curves, etc. Once you're satisfied, you have a production model.

Once you have that model, you can deploy it, for example, to a mobile device using TensorFlow Lite ([Figure 19-3](#)).

TensorFlow Serving fits into this architecture by providing the infrastructure for hosting your model on a server. Clients can then use HTTP to pass requests to this server along with a data payload. The data will be passed to the model, which will run inference, get the results, and return them to the client ([Figure 19-4](#)).

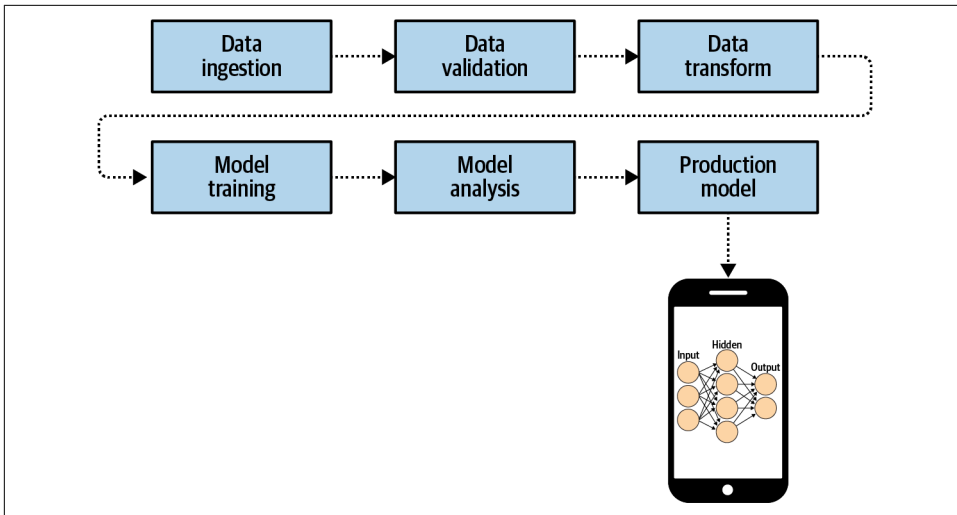


Figure 19-3. Deploying your production model to mobile

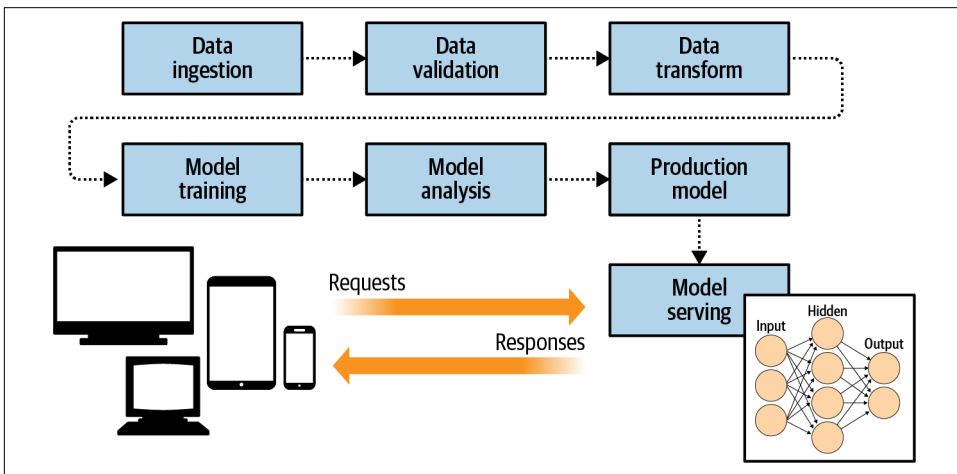


Figure 19-4. Adding model serving architecture to the pipeline

An important artifact of this type of architecture is that you can also control the versioning of the models used by your clients. When models are deployed to mobile devices, for example, you can end up with model drift, where different clients have different versions. But when serving from an infrastructure, as in [Figure 19-4](#), you can avoid this. Additionally, this makes it possible to experiment with different model versions, where some clients will get the inference from one version of the model, while others get it from other versions ([Figure 19-5](#)).

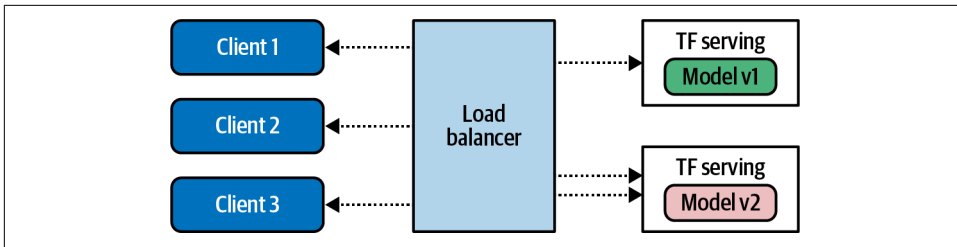


Figure 19-5. Using TensorFlow Serving to handle multiple model versions

Installing TensorFlow Serving

TensorFlow Serving can be installed with two different server architectures. The first, `tensorflow-model-server`, is a fully optimized server that uses platform-specific compiler options for various architectures. In general it's the preferred option, unless your server machine doesn't have those architectures. The alternative, `tensorflow-model-server-universal`, is compiled with basic optimizations that should work on all machines, and provides a nice backup if `tensorflow-model-server` does not work. There are several methods by which you can install TensorFlow Serving, including using Docker or a direct package installation using `apt`. We'll look at both of those options next.

Installing Using Docker

Using Docker is perhaps the easiest way to get up and running quickly. To get started, use `docker pull` to get the TensorFlow Serving package:

```
docker pull tensorflow/serving
```

Once you've done this, clone the TensorFlow Serving code from GitHub:

```
git clone https://github.com/tensorflow/serving
```

This includes some sample models, including one called Half Plus Two that, given a value, will return half that value plus two. To do this, first set up a variable called `TESTDATA` that contains the path of the sample models:

```
TESTDATA="$(pwd)/serving/tensorflow_serving/servables/tensorflow/testdata"
```

You can now run TensorFlow Serving from the Docker image:

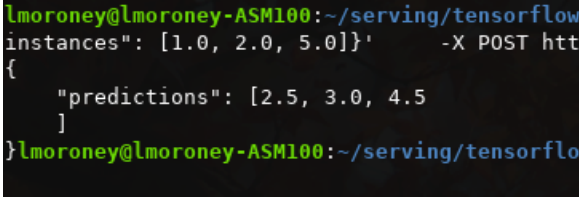
```
docker run -t --rm -p 8501:8501 \
  -v "$TESTDATA/saved_model_half_plus_two_cpu:/models/half_plus_two" \
  -e MODEL_NAME=half_plus_two \
  tensorflow/serving &
```

This will instantiate a server on port 8501—you’ll see how to do that in more detail later in this chapter—and execute the model on that server. You can then access the model at `http://localhost:8501/v1/models/half_plus_two:predict`.

To pass the data that you want to run inference on, you can POST a tensor containing the values to this URL. Here’s an example using `curl` (run this in a separate terminal if you’re running on your development machine):

```
curl -d '{"instances": [1.0, 2.0, 5.0]}' \
-X POST http://localhost:8501/v1/models/half_plus_two:predict
```

You can see the results in [Figure 19-6](#).



```
lmoroney@lmoroney-ASM100:~/serving/tensorflow
instances": [1.0, 2.0, 5.0]}' -X POST htt
{
  "predictions": [2.5, 3.0, 4.5
]
}
lmoroney@lmoroney-ASM100:~/serving/tensorflo
```

Figure 19-6. Results of running TensorFlow Serving

While the Docker image is certainly convenient, you might also want the full control of installing it directly on your machine. You’ll explore how to do that next.

Installing Directly on Linux

Whether you are using `tensorflow-model-server` or `tensorflow-model-server-universal`, the package name is the same. So, it’s a good idea to remove `tensorflow-model-server` before you start so you can ensure you get the right one. If you want to try this on your own hardware, I’ve provided a [Colab notebook](#) in the GitHub repo with the code:

```
apt-get remove tensorflow-model-server
```

Then add the [TensorFlow package source](#) to your system:

```
echo "deb http://storage.googleapis.com/tensorflow-serving-apt stable
tensorflow-model-server tensorflow-model-server-universal" | tee
/etc/apt/sources.list.d/tensorflow-serving.list && \ curl
https://storage.googleapis.com/tensorflow-serving-apt/tensorflow-
serving.release.pub.gpg | apt-key add -
```

If you need to use `sudo` on your local system, you can do so like this:

```
sudo echo "deb http://storage.googleapis.com/tensorflow-serving-apt stable
tensorflow-model-server tensorflow-model-server-universal" | sudo tee
/etc/apt/sources.list.d/tensorflow-serving.list && \ curl
https://storage.googleapis.com/tensorflow-serving-apt/tensorflow-
serving.release.pub.gpg | sudo apt-key add -
```

You'll need to update `apt-get` next:

```
apt-get update
```

Once this has been done, you can install the model server with `apt`:

```
apt-get install tensorflow-model-server
```

And you can ensure you have the latest version by using the following:

```
apt-get upgrade tensorflow-model-server
```

The package should now be ready to use.

Building and Serving a Model

In this section we'll do a walkthrough of the complete process of creating a model, preparing it for serving, deploying it with TensorFlow Serving, and then running inference using it.

You'll use the simple "Hello World" model that we've been exploring throughout the book:

```
xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])

model.compile(optimizer='sgd', loss='mean_squared_error')

history = model.fit(xs, ys, epochs=500, verbose=0)

print("Finished training the model")

print(model.predict([10.0]))
```

This should train very quickly and give you a result of 18.98 or so, when asked to predict Y when X is 10.0.

Next, the model needs to be saved. You'll need a temporary folder to save it in:

```
import tempfile
import os
MODEL_DIR = tempfile.gettempdir()
version = 1
export_path = os.path.join(MODEL_DIR, str(version))
print(export_path)
```

When running in Colab, this should give you output like `/tmp/1`. If you're running on your own system you can export it to whatever directory you want, but I like to use a temp directory.

If there's anything in the directory you're saving the model to, it's a good idea to delete it before proceeding (avoiding this issue is one reason why I like using a temp directory!). To ensure that your model is the master, you can delete the contents of the `export_path` directory:

```
if os.path.isdir(export_path):
    print('\nAlready saved a model, cleaning up\n')
    !rm -r {export_path}
```

Now you can save the model:

```
model.save(export_path, save_format="tf")

print('\nexport_path = {}'.format(export_path))
!ls -l {export_path}
```

Once this is done, take a look at the contents of the directory. The listing should show something like this:

```
INFO:tensorflow:Assets written to: /tmp/1/assets

export_path = /tmp/1
total 48
drwxr-xr-x 2 root root 4096 May 21 14:40 assets
-rw-r--r-- 1 root root 39128 May 21 14:50 saved_model.pb
drwxr-xr-x 2 root root 4096 May 21 14:50 variables
```

The TensorFlow Serving tools include a utility called `saved_model_cli` that can be used to inspect a model. You can call this with the `show` command, giving it the directory of the model in order to get the full model metadata:

```
!saved_model_cli show --dir {export_path} --all
```

Note that the `!` is used for Colab to indicate a shell command. If you're using your own machine, it isn't necessary.

The output of this command will be very long, but will contain details like this:

```
signature_def['serving_default']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['dense_input'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 1)
      name: serving_default_dense_input:0
  The given SavedModel SignatureDef contains the following output(s):
    outputs['dense'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 1)
      name: StatefulPartitionedCall:0
```

Note the content of the `signature_def`, which in this case is `serving_default`. You'll need them later.

Also note that the inputs and outputs have a defined shape and type. In this case, each is a float and has the shape $(-1, 1)$. You can effectively ignore the -1 , and just bear in mind that the input to the model is a float and the output is a float.

If you are using Colab, you need to tell the operating system where the model directory is so that when you run TensorFlow Serving from a bash command, it will know the location. This can be done with an environment variable in the operating system:

```
os.environ["MODEL_DIR"] = MODEL_DIR
```

To run the TensorFlow model server with a command line, you need a number of parameters. First, you'll use the `--bg` switch to ensure that the command runs in the background. The `nohup` command stands for “no hangup,” requesting that the script continues to run. Then you need to specify a couple of parameters to the `tensorflow_model_server` command. `rest_api_port` is the port number you want to run the server on. Here, it's set to 8501. You then give the model a name with the `model_name` switch—here I've called it `helloworld`. Finally, you then pass the server the path to the model you saved in the `MODEL_DIR` operating system environment variable with `model_base_path`. Here's the code:

```
%bash --bg
nohup tensorflow_model_server \
  --rest_api_port=8501 \
  --model_name=helloworld \
  --model_base_path="{MODEL_DIR}" >server.log 2>&1
```

At the end of the script, there is code to output the results to `server.log`. The output of this in Colab will simply be this:

```
Starting job # 0 in a separate thread.
```

You can inspect it with the following:

```
!tail server.log
```

Examine this output, and you should see that the server started successfully with a note showing that it is exporting the HTTP/REST API at `localhost:8501`:

```
2020-05-21 14:41:20.026123: I tensorflow_serving/model_servers/server.cc:358]
Running gRPC ModelServer at 0.0.0.0:8500 ...
[warn] getaddrinfo: address family for nodename not supported
2020-05-21 14:41:20.026777: I tensorflow_serving/model_servers/server.cc:378]
Exporting HTTP/REST API at:localhost:8501 ...
[evhttp_server.cc : 238] NET_LOG: Entering the event loop ...
```

If it fails, you should see a notification about the failure. Should that happen, you might need to restart your system.

If you want to test the server, you can do so within Python:

```
import json
xs = np.array([[9.0], [10.0]])
data = json.dumps({"signature_name": "serving_default", "instances":
                  xs.tolist()})
print(data)
```

To send data to the server, you need to get it into JSON format. So with Python it's a case of creating a Numpy array of the values you want to send—in this case it's a list of two values, 9.0 and 10.0. Each of these is an array in itself, because, as you saw earlier, the input shape is $(-1,1)$. Single values should be sent to the model, so if you want multiple ones, it should be a list of lists, with the inner lists having single values only.

Use `json.dumps` in Python to create the payload, which is two name/value pairs. The first is the signature name to call on the model, which in this case is `serving_default` (as you'll recall from earlier, when you inspected the model). The second is `instances`, which is the list of values you want to pass to the model.

Printing this will show you what the payload looks like:

```
{"signature_name": "serving_default", "instances": [[9.0], [10.0]]}
```

You can call the server using the `requests` library to do an HTTP POST. Note the URL structure. The model is called `helloworld`, and you want to run its prediction. The POST command requires data, which is the payload you just created, and a headers specification, where you're telling the server the content type is JSON:

```
import requests
headers = {"content-type": "application/json"}
json_response =
    requests.post('http://localhost:8501/v1/models/helloworld:predict',
                  data=data, headers=headers)

print(json_response.text)
```

The response will be a JSON payload containing the predictions:

```
{
  "predictions": [[16.9834747], [18.9806728]]
}
```

Exploring Server Configuration

In the preceding example, you created a model and served it by launching TensorFlow Serving from a command line. You used parameters to determine which model to serve, and provide metadata such as which port it should be served on. TensorFlow Serving gives you more advanced serving options via a configuration file.

The model configuration file adheres to a protobuf format called `ModelServerConfig`. The most commonly used setting within this file is `model_config_list`, which

contains a number of configurations. This allows you to have multiple models, each served at a particular name. So, for example, instead of specifying the model name and path when you launch TensorFlow Serving, you can specify them in the config file like this:

```
model_config_list {
  config {
    name: '2x-1model'
    base_path: '/tmp/2xminus1/'
  }
  config {
    name: '3x+1model'
    base_path: '/tmp/3xplus1/'
  }
}
```

If you now launch TensorFlow Serving with this configuration file instead of using switches for the model name and path, you can map multiple URLs to multiple models. For example, this command:

```
%%bash --bg
nohup tensorflow_model_server \
  --rest_api_port=8501 \
  --model_config=/path/to/model.config >server.log 2>&1
```

will now allow you to POST to `<server>:8501/v1/models/2x-1model:predict` or `<server>:8501/v1/models/3x+1model:predict`, and TensorFlow Serving will handle loading the correct model, performing the inference, and returning the results.

The model configuration can also allow you to specify versioning details per model. So, for example, if you update the previous model configuration to this:

```
model_config_list {
  config {
    name: '2x-1model'
    base_path: '/tmp/2xminus1/'
    model_version_policy: {
      specific {
        versions : 1
        versions : 2
      }
    }
  }
  config {
    name: '3x+1model'
    base_path: '/tmp/3xplus1/'
    model_version_policy: {
      all : {}
    }
  }
}
```

this will allow you to serve versions 1 and 2 of the first model, and all versions of the second model. If you don't use these settings, the one that's configured in the `base_path` or, if that isn't specified, the latest version of the model will be served. Additionally, the specific versions of the first model can be given explicit names, so that, for example, you could designate version 1 to be your master and version 2 your beta by assigning these labels. Here's the updated configuration to implement this:

```
model_config_list {
  config {
    name: '2x-1model'
    base_path: '/tmp/2xminus1/'
    model_version_policy: {
      specific {
        versions : 1
        versions : 2
      }
    }
    version_labels {
      key: 'master'
      value: 1
    }
    version_labels {
      key: 'beta'
      value: 2
    }
  }
  config {
    name: '3x+1model'
    base_path: '/tmp/3xplus1/'
    model_version_policy: {
      all : {}
    }
  }
}
```

Now, if you want to access the beta version of the first model, you can do so as follows:

```
<server>:8501/v1/models/2x-1model/versions/beta
```

If you want to change your model server configuration without stopping and restarting the server, you can have it poll the configuration file periodically; if it sees a change, you'll get the new configuration. So, for example, say you don't want the master to be version 1 anymore, and you instead want it to be v2. You can update the configuration file to take account of this change, and if the server has been started with the `--model_config_file_poll_wait_seconds` parameter, as shown here, once that timeout is hit, the new configuration will be loaded:


```
%%bash --bg
nohup tensorflow_model_server \
  --rest_api_port=8501 \
  --model_config=/path/to/model.config
  --model_config_file_poll_wait_seconds=60 >server.log 2>&1
```

Summary

In this chapter you had your first look at TFX. You saw that any machine learning system has components that go far beyond just building a model, and learned how one of those components—TensorFlow Serving, which provides model serving capabilities—can be installed and configured. You explored building a model, preparing it for serving, deploying it to a server, and then running inference using an HTTP POST request. After that you looked into the options for configuring your server with a configuration file, examining how to use it to deploy multiple models and different versions of those models. In the next chapter we'll go in a different direction and see how distributed models can be managed for distributed learning while maintaining a user's privacy with federated learning.

AI Ethics, Fairness, and Privacy

In this book you've taken a programmer's tour of the APIs available in the TensorFlow ecosystem to train models for a variety of tasks and deploy those models to a number of different surfaces. It's this methodology of *training* models, using labeled data instead of explicitly programming logic yourself, that is at the heart of the machine learning and, by extension, artificial intelligence revolution.

In [Chapter 1](#), we condensed the changes this involves for a programmer into a diagram, shown in [Figure 20-1](#).

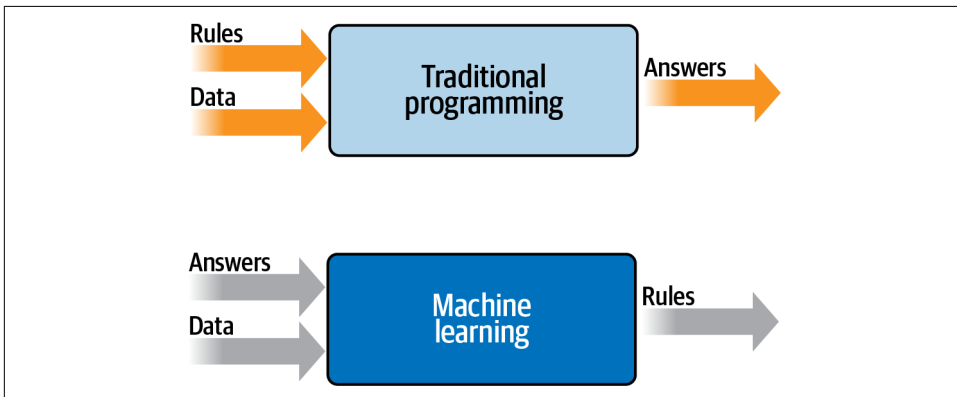


Figure 20-1. Traditional programming versus machine learning

This leads to a new challenge. With source code, it's possible to inspect how a system works by stepping through and exploring the code. But when you build a model, even a simple one, the outcome is a binary file consisting of the learned parameters within the model. These can be weights, biases, learned filters, and more. As a result, they

can be quite obscure, leading to difficulty in interpreting what they do and how they work.

And if we, as a society, begin to depend on trained models to help us with computing tasks, it's important for us to have some level of transparency regarding how the models work—so it's important for you, as an AI engineer, to understand building for ethics, fairness, and privacy. There's enough to learn to fill several books, so in this chapter we'll really only be scraping the surface, but I hope it's a good introduction to helping you learn what you need to know.

Most importantly, building systems with a view to being fair to users isn't a new thing, nor is it virtue signaling or political correctness. Regardless of anybody's feelings about the importance of engineering for overall fairness, there's one indisputable fact that I aim to demonstrate in this chapter: that building systems with a view to being both *fair* and *ethical* is the right thing to do from an engineering perspective and will help you avoid future technical debt.

Fairness in Programming

While recent advances in machine learning and AI have brought the concepts of ethics and fairness into the spotlight, it's important to note that disparity and unfairness have always been topics of concern in computer systems. In my career, I have seen many examples where a system has been engineered for one scenario without considering the overall impact with regard to fairness and bias.

Consider this example: your company has a database of its customers and wants to launch a marketing campaign to target more customers in a particular zip code where it has identified a growth opportunity. To do so, the company will send discount coupons to people in that zip code with whom it has connected, but who haven't yet purchased anything. You could write SQL like this to identify these potential customers:

```
SELECT * FROM Customers WHERE ZIP=target_zip AND PURCHASES=0
```

This might seem to be perfectly sensible code. But consider the demographics at that zip code. What if the majority of people who live there are of a particular race or age? Instead of growing your customer base evenly, you could be overtargeting one segment of the population, or worse, discriminating against another by offering discounts to people of one race but not another. Over time, continually targeting like this could result in a customer base that is skewed against the demographics of society, ultimately painting your company into a corner of primarily serving one segment of society.

Here's another example—and this one really happened to me! Back in [Chapter 1](#), I used a few emoji to demonstrate the concept of machine learning for activity detection (see [Figure 20-2](#)).

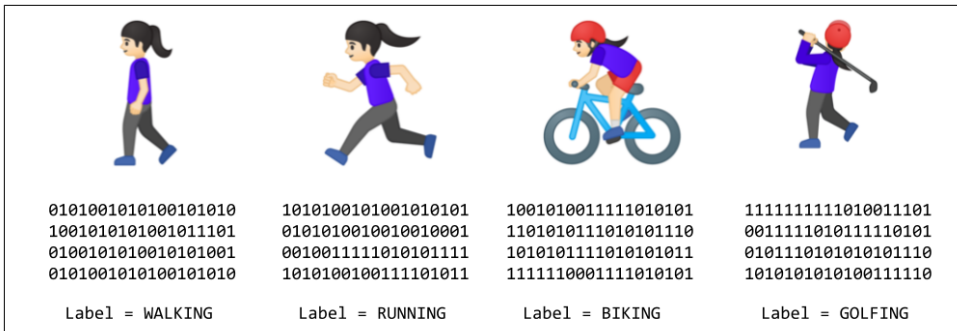


Figure 20-2. Emoji to demonstrate machine learning

There's a story behind these that started several years ago. I was fortunate enough to visit Tokyo, and at the time I was beginning to learn how to run. A good friend of mine in the city invited me to run around the Imperial Palace. She sent a text with a couple of emoji in it that looked like Figure 20-3.

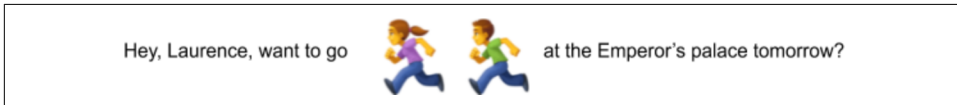


Figure 20-3. Text containing emoji

The text contained two emoji, a woman running and a man running. I wanted to reply and send the same emoji, but was doing it from a desktop chat application, which didn't have the modern ability to pick an emoji from a list. If you wanted an emoji, you had to type a shortcode.

When I typed the shortcode (running), I would get the male emoji. But if I wanted the female emoji, there seemed to be no way of doing it. After a bit of Googling, I found that I could get the female emoji by typing (running)+♀. Then the question became, how do you type ♀?

It depends on the operating system, but for example, on Windows you have to hold Alt and type 12 on the numeric keypad. On Linux you have to press Left Ctrl-Shift-U and then type the Unicode for the symbol, which is 2640.

That's a lot of work to get a female emoji, not to mention that the implicit declaration of a female emoji is a male one that gets modified to make it female with the addition of ♀. This is not inclusive programming. But how did it arise?

Consider the history of emoji. When they were first used, they were simply characters in text that were typed with a sideways view, like :) for smiling or ;) for winking, or my personal favorite *:)) which looks like Ernie from *Sesame Street*. They're inherently genderless because they're so low-resolution. As emoji (or emoticons) evolved from

characters to graphics, they were typically monochrome “stick man”-type illustrations. The clue is in the name—stick *man*. As graphics became better, particularly on mobile devices, the emoji then became clearer. For example, in the original iPhone OS (2.2), the running emoji (renamed “Person Running”) looked like this: 🏃.

As graphics improved further and screen pixel densities increased, the emoji continued to evolve, and by iOS 13.3 it looked like [Figure 20-4](#).



Figure 20-4. Person Running emoji from iOS 13.3

When it comes to engineering, as well as improving the graphics, it’s important to maintain *backward compatibility*—so if earlier versions of your software used the shortcode (running) to indicate a stick man running, later versions with richer graphics can give you a graphic like this one, which is now very clearly a man running.

Consider what this would look like in pseudocode:

```
if shortcode.contains("(running)"){
    showGraphic(personRunning)
}
```

This is well-engineered code because it maintains backward compatibility as the graphic changes. You never need to update your code for new screens and new graphics; you just change the `personRunning` resource. But the *effect* changes for your end users, so you then identify that you also need to have a Woman Running emoji to be fair.

You can’t use the same shortcode, however, and you don’t want to break backward compatibility, so you have to amend your code, maybe to something like this:

```
if shortcode.contains("(running)"){
    if(shortcode.contains("+♀️")){
        showGraphic(womanRunning);
    } else {
        showGraphic(personRunning);
    }
}
```

From a coding perspective, this makes sense. You can provide the additional functionality without breaking backward compatibility, and it’s easy to remember—if you want a female runner you use a female Venus symbol. But life isn’t just from a coding

perspective, as this example shows. Engineering like this led to a runtime environment that adds excess friction to the use cases of a significant portion of the population.

The technical debt incurred from the beginning, when gender equality wasn't considered when creating emoji, still lingers to this day in workarounds like this. Check the [Woman Running page](#) on Emojipedia, and you'll see that this emoji is defined as a zero-width joiner (ZWJ) sequence that combines Person Running, a ZWJ, and a Venus symbol. The only way to try to give end users the proper experience of having a female running emoji is to implement a workaround.

Fortunately, as many apps that offer emoji now use a selector where you choose it from a menu, as opposed to typing a shortcode, the issue has become somewhat hidden. But it's still there under the surface, and while this example is rather trivial, I hope it demonstrates how past decisions that didn't consider fairness or bias can have ramifications further down the line. Don't sacrifice your future to save your present!

So, back to AI and machine learning. As we're at the dawn of a new age of application types, it's vitally important for you to consider all aspects of the use of your application. You want to ensure that fairness is built in as much as possible. You also want to ensure that bias is avoided as much as possible. It's the right thing to do, and it can help avoid the technical debt of future workarounds.

Fairness in Machine Learning

Machine learning systems are data-driven, not code-driven, so identifying biases and other problematic areas becomes an issue of understanding your data. It requires tooling to help you explore your data and see how it flows through a model. Even if you have excellent data, a poorly engineered system could lead to issues. The following are some tips to consider when building ML systems that can help you avoid such issues:

Determine if ML is actually necessary

It might go without saying, but, as new trends hit the technology market, there's always pressure to implement them. There's often a push from investors, or from sales channels or elsewhere, to show that you are cutting-edge and using the latest and greatest. As such, you may be given the requirement to incorporate machine learning into your product. But what if it isn't necessary? What if, in order to hit this nonfunctional requirement, you paint yourself into a corner because ML isn't suited to the task, or because while it might be useful in the future, you don't have adequate data coverage right now?

I once attended a student competition where the participants took on the challenge of image generation using generative adversarial networks (GANs) to predict what the lower half of a face looks like based on the upper half of the face.

This was during a flu season before COVID-19, and, in a classic example of Japanese grace, many people were wearing face masks. The idea was to see if one could predict the face below the mask. For this task they needed access to facial data, so they used the IMDb dataset of **face images with age and gender labels**. The problem? Given that the source is IMDb, the vast majority of the faces in this dataset are not Japanese. As such, their model did a great job of predicting *my* face, but not their own. In the rush to produce an ML solution when there wasn't adequate data coverage, the students produced a biased solution. This was just a show-and-tell competition, and their work was brilliant, but it was a great reminder that rushing to market with an ML product when one isn't necessarily needed, or when there isn't sufficient data to build a proper model, can lead you down the road of building biased models and incurring heavy future technical debt.

Design and implement metrics from day one

Or maybe that should read from day zero, as we're all programmers here. In particular, if you are amending or upgrading a non-ML system to add ML, you should do as much as you can to track how your system is currently being used. Consider the emoji story from earlier as an example. If people had identified early on—given there are as many female runners as male runners—that having a male Person Running emoji was a user experience mistake, then a problem might never have arisen at all. You should always try to understand your users' needs so that as you design a data-oriented architecture for ML, you can ensure that you have adequate coverage to meet these needs, and possibly predict future trends and get ahead of them.

Build a minimum viable model and iterate

You should experiment with building a minimum viable model *before* you set any expectations about deploying ML models into your system. ML and AI are not a magic solution for everything. Given the data at hand, build a minimum viable product (MVP) that gets you on the road to having ML in your system. Does it do the job? Do you have a pathway to gathering more of the data needed to extend the system while keeping it fair for all users? Once you have your MVP, iterate, prototype, and continue to test before rushing something to production.

Ensure your infrastructure supports rapid redeployment

Whether you are deploying your model to a server with TensorFlow Serving, to mobile devices with TensorFlow Lite, or to browsers with TensorFlow.js, it's important to keep an eye on how you can *redploy* the model if needed. If you hit a scenario where it is failing (for any reason, not just bias), it's good to have the ability to rapidly deploy a new model without breaking your end users' experience. Using a configuration file with TensorFlow Serving, for example, allows you to define multiple models with named values that you can use to rapidly switch between them. With TensorFlow Lite, your model is deployed as an asset, so

instead of hardcoding that into your app, you might want the app to check the internet for updated versions of the model and update if it detects one. In addition, abstracting the code that runs inference with a model—avoiding hard-coded labels, for example—can help you to avoid regression errors when you redeploy.

Tools for Fairness

There's a growing market for tooling for understanding the data used to train models, the models themselves, and the inference output of the models. We'll explore a few of the currently available options here.

The What-If Tool

One of my favorites is the What-If Tool from Google. Its aim is to let you inspect an ML model with minimal coding required. With this tool, you can inspect the data and the output of the model for that data together. It has a [walkthrough](#) that uses a model based on about 30,000 records from the 1994 US Census dataset that is trained to predict what a person's income might be. Imagine, for example, that this is used by a mortgage company to determine whether a person may be able to pay back a loan, and thus to determine whether or not to grant them the loan.

One part of the tool allows you to select an inference value and see the data points from the dataset that led to that inference. For example, consider [Figure 20-5](#).

This model returns a probability of low income from 0 to 1, with values below 0.5 indicating high income and those above 0.5 low income. This user had a score of 0.528, and in our hypothetical mortgage application scenario could be rejected as having too low an income. With the tool, you can actually change some of the user's data—for example, their age—and see what the effect on the inference would be. In the case of this person, changing their age from 42 to 48 gave them a score on the other side of the 0.5 threshold, and as a result changed them from being a “reject” on the loan application to an “accept.” Note that nothing else about the user was changed—just their age. This gives a strong signal that there's a potential age bias in the model.

The What-If Tool allows you to experiment with various signals like this, including details like gender, race, and more. To prevent a one-off situation being the tail that wags the dog, causing you to change your entire model to prevent an issue that lies with one customer and not the model itself, the tool includes the ability to find the nearest counterfactuals. That is, it finds the closest set of data that results in a different inference so you can start to dive into your data (or model architecture) in order to find biases.

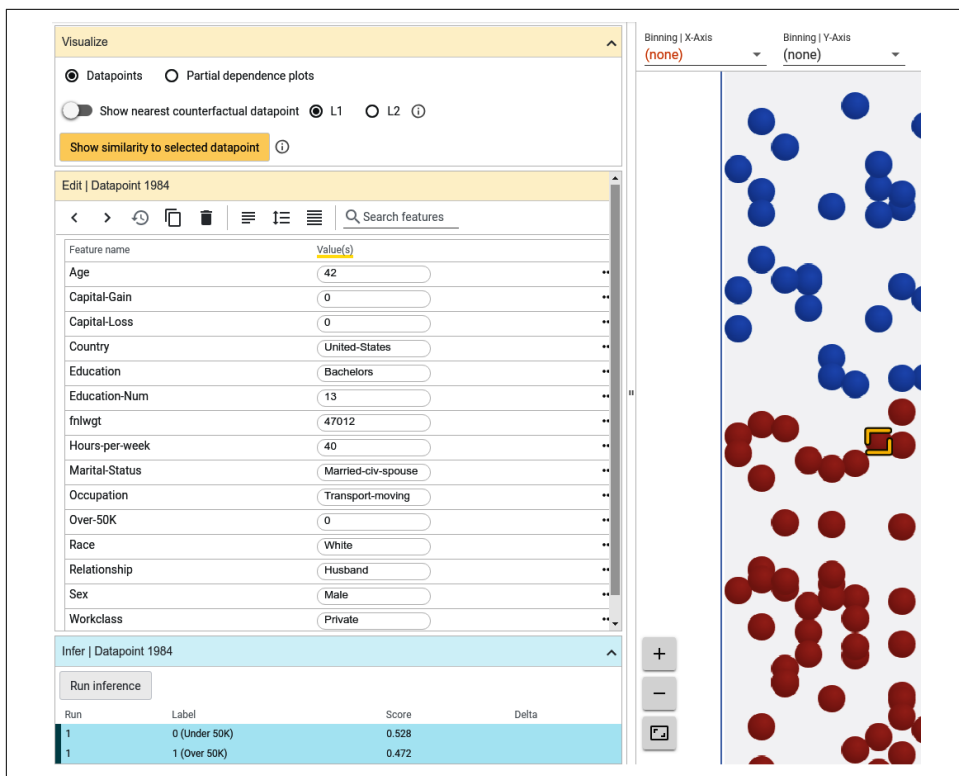


Figure 20-5. Using the What-If Tool

I’m just touching the surface of what the What-If Tool can do here, but I’d strongly recommend checking it out. There are lots of [examples](#) of what you can do with it on the site. At its core—as the name suggests—it gives you tools to test “what-if” scenarios before you deploy. As such, I believe it can be an essential part of your ML toolbox.

Facets

Facets is a tool that can complement the What-If Tool to give you a deep dive into your data through visualizations. The goal of Facets is to help you understand the distribution of values across features in your dataset. It’s particularly useful if your data is split into multiple subsets for training, testing, validation, or other uses. In such cases, you can easily end up in a situation where data in one split is skewed in favor of a particular feature, leading you to have a faulty model. This tool can help you determine whether you have sufficient coverage of each feature for each split.

For example, using the same US Census dataset as in the previous example with the What-If Tool, a little examination shows that the training/test splits are very good,

but use of the capital gain and capital loss features might have a skewing effect on the training. Note in [Figure 20-6](#), when inspecting quantiles, that the large crosses are very well balanced across all of the features except these two. This indicates that the majority of the data points for these values are zeros, but there are a few values in the dataset that are much higher. In the case of capital gain, you can see that 91.67% of the training set is zeros, with the other values being close to 100,000. This might skew your training, and can be seen as a debugging signal. This could introduce a bias in favor of a very small part of your population.

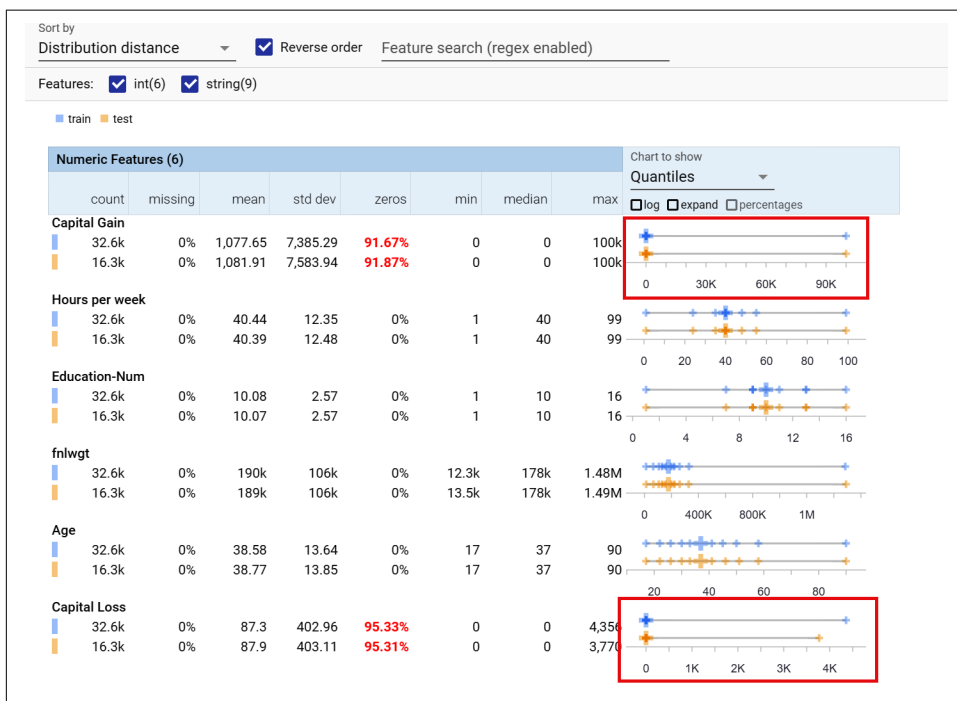


Figure 20-6. Using Facets to explore a dataset

Facets also includes a tool called Facets Dive that lets you visualize the contents of your dataset according to a number of axes. It can help identify errors in your dataset, or even preexisting biases so you know how to handle them. For example, consider [Figure 20-7](#), where I split the dataset by target, education level, and gender.

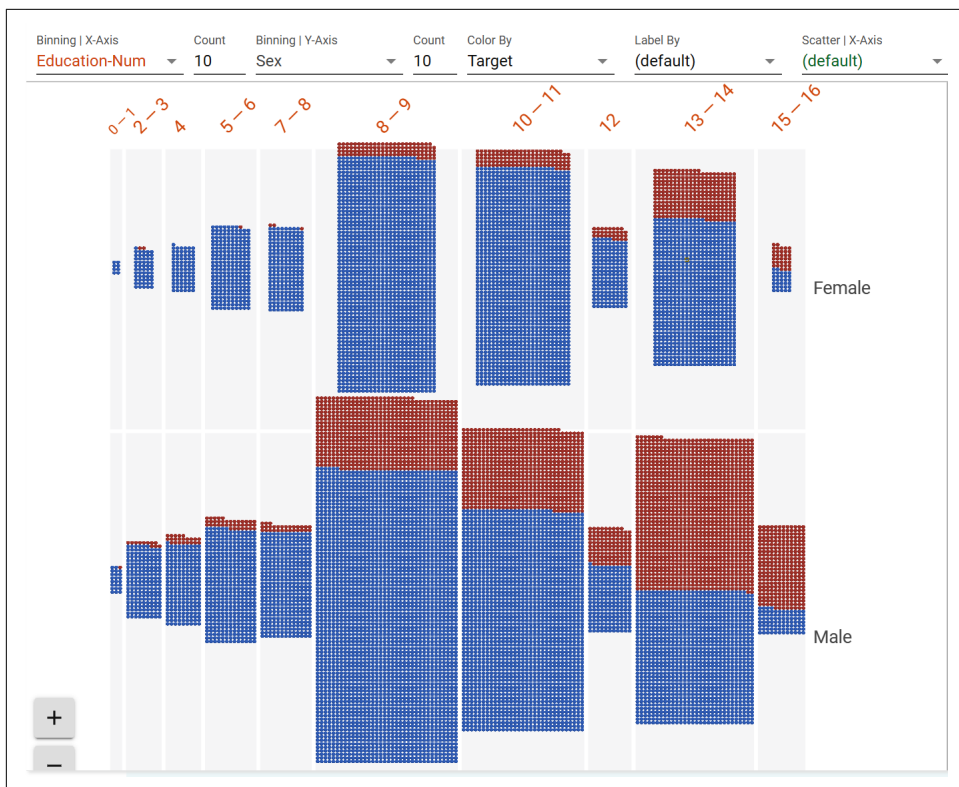


Figure 20-7. A deep dive with Facets

Red means “high income predicted,” and left to right are levels of education. In almost every case the probability of a male having a high income is greater than a female, and, in particular, with higher levels of education the contrast becomes stark. Look, for example, at the 13–14 column (which is the equivalent of a bachelor’s degree): the data shows a far higher percentage of men being high earners than women with the same education level. While there are many other factors in the model to determine earning level, having such a disparity for highly educated people is a likely indicator of bias in the model.

To help you identify features such as these, along with the What-If Tool, I strongly recommend using Facets to explore your data and your model’s output.

Both of these tools come from the People + AI Research (PAIR) team at Google. I’d recommend you bookmark [their site](#) to keep an eye on the latest releases, as well as the [People + AI Guidebook](#) to help you follow a human-centered approach to AI.

Federated Learning

After your models are deployed and distributed, there's a huge opportunity for them to be continually improved, based on how they are used by your entire user base. On-device keyboards with predictive text, for example, must learn from every user to be effective. But there's a catch—in order for a model to learn, the data needs to be collected, and gathering data from end users to train a model, particularly without their consent, can be a massive invasion of privacy. It's not right for every word your end users type to be used to improve keyboard predictions, because then the contents of every email, every text, every message would be known to an outside third party. So, to allow for this type of learning, a technique to maintain the user's privacy, while also sharing the valuable parts of the data, needs to be used. This is commonly called *federated learning*, and we'll explore it in this section.

The core idea behind federated learning is that user data is *never* sent to a central server. Instead, a procedure like the one outlined in the following sections is used.

Step 1. Identify Available Devices for Training

First of all, you'll need to identify a set of your users who are suitable for training work. It's important to consider the impact on the user performing on-device training. To determine whether the device is available, weigh factors such as whether the device is already in use, or whether it is plugged into a power supply (see [Figure 20-8](#)).

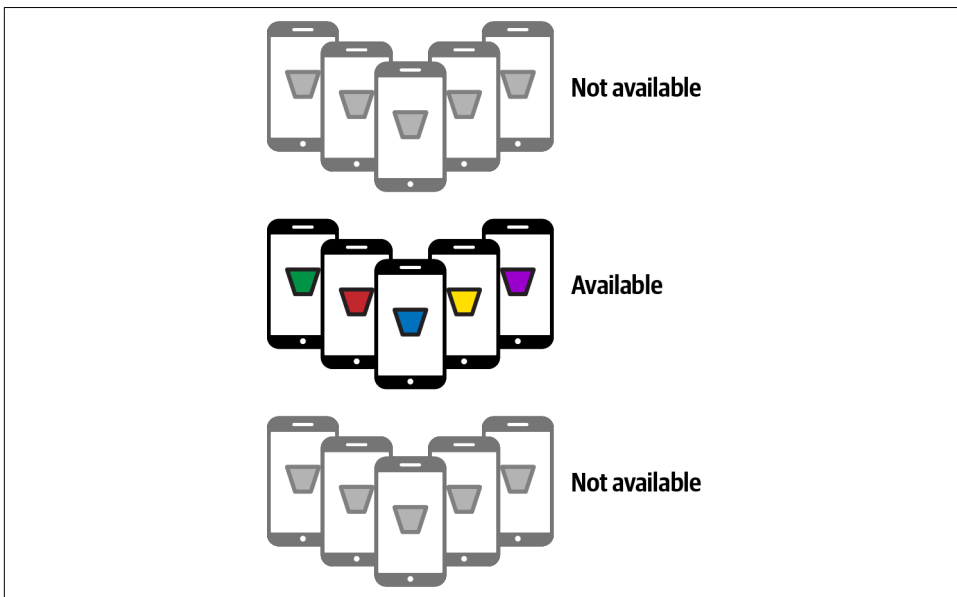


Figure 20-8. Identifying available devices

Step 2. Identify Suitable Available Devices for Training

Of the available ones, not all will be suitable. They may not have sufficient data, they may not have been used recently, etc. There are a number of factors that could determine suitability, based on your training criteria. Based on these, you'll have to filter the available devices down into a set of *suitable* available devices (Figure 20-9).

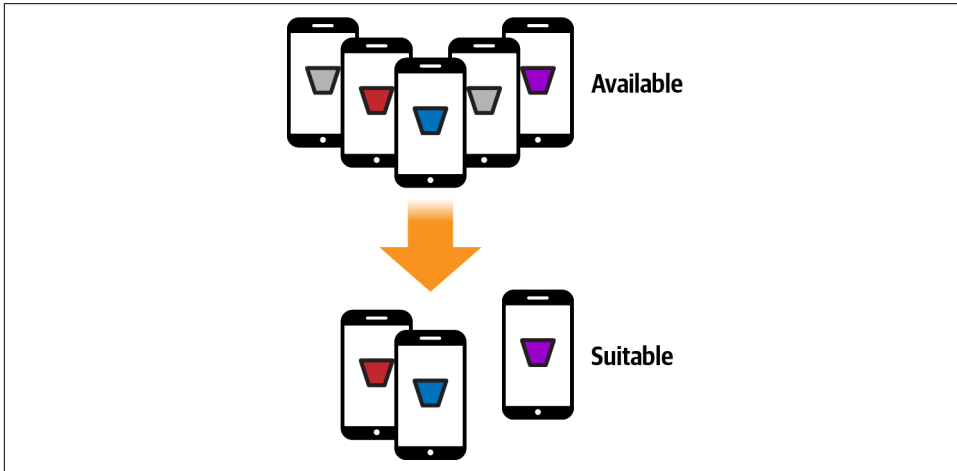


Figure 20-9. Choosing suitable available devices

Step 3. Deploy a Trainable Model to Your Training Set

Now that you've identified a set of suitable available devices, you can deploy a model to them (Figure 20-10). The model will be trained *on the devices*, which is why devices that are not currently in use and are plugged in (to avoid draining the battery) are the suitable family to use. *Note that there is no public API to do on-device training with TensorFlow at this time.* You can test this environment in Colab, but there's no Android/iOS equivalent at the time of writing.

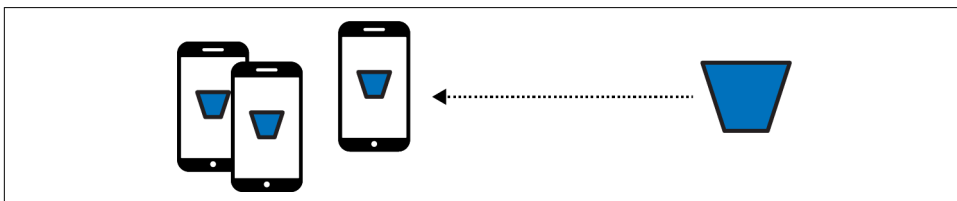


Figure 20-10. Deploying a new training model to the devices

Step 4. Return the Results of the Training to the Server

Note that the *data* used to train the model on the individual's device *never* leaves the device. However weights, biases, and other parameters learned by the model can leave the device. Another level of security and privacy can be added here (discussed in “[Secure Aggregation with Federated Learning](#)” on page 352). In this case, the values learned by each of the devices can be passed to the server, which can then aggregate them back into the master model, effectively creating a new version of the model with the distributed learning of each of the clients ([Figure 20-11](#)).

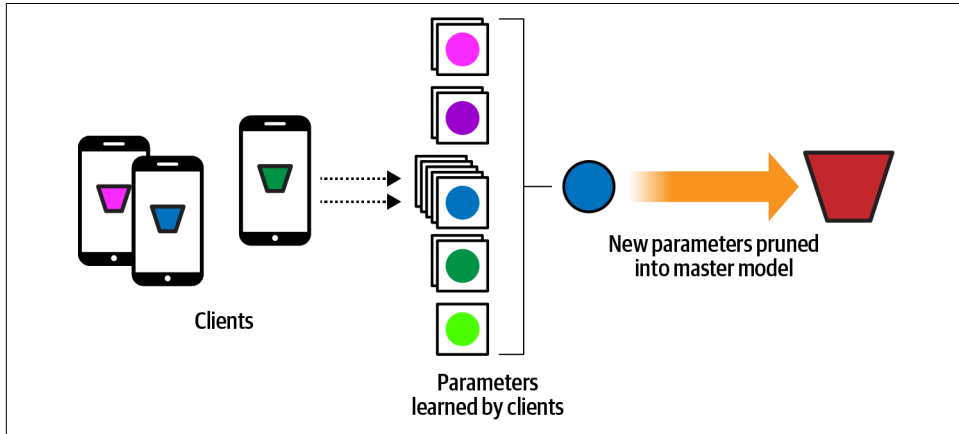


Figure 20-11. Creating a new master model from the learnings of the clients

Step 5. Deploy the New Master Model to the Clients

Then, as clients become available to receive the new master model, it can get deployed to them, so everyone can access the new functionality ([Figure 20-12](#)).

Following this pattern will allow you to have a conceptual framework where you have a centralized model that can be trained from the experiences of all of your users, without violating their privacy by sending data to your server. Instead, a subset of the training is done directly on their devices, and the *results* of that training are all that ever leaves the device. As described next, a method called *secure aggregation* can be used to provide an additional layer of privacy through obfuscation.

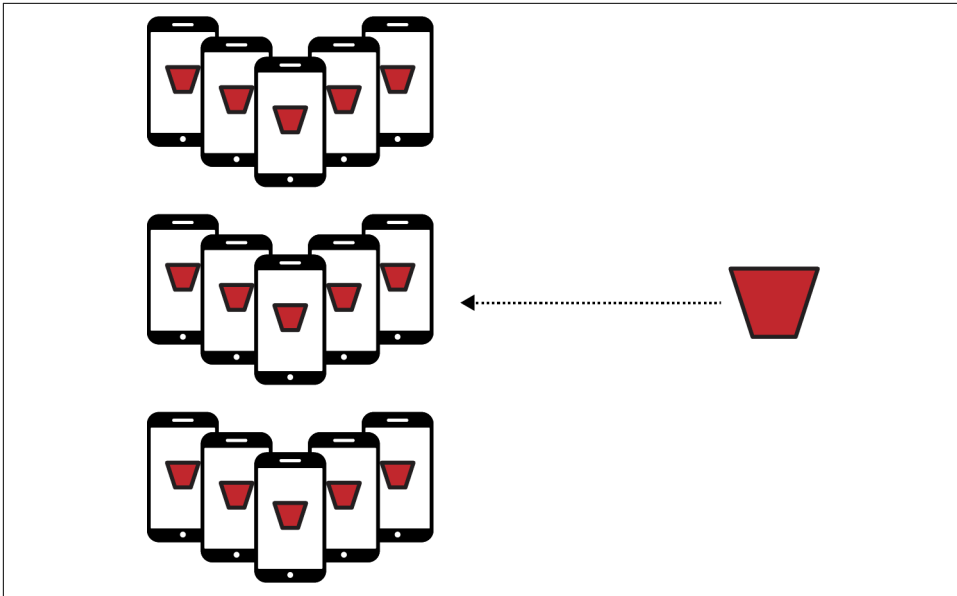


Figure 20-12. The new master model gets deployed to all of the clients

Secure Aggregation with Federated Learning

The previous walkthrough demonstrated the conceptual framework of federated learning. This can be combined with the concept of secure aggregation to further obfuscate the learned weights and biases while in transit from the client to the server. The idea behind it is simple. The server pairs up devices with others in a buddy system. For example, consider [Figure 20-13](#), where there are a number of devices, each of which is given two buddies. Each buddy pair is sent the same random value to be used as a multiplier to obfuscate the data it sends.

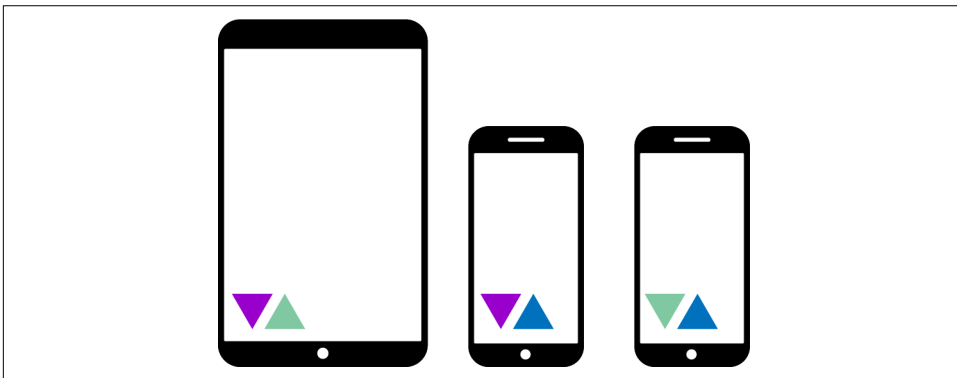


Figure 20-13. Buddy devices

Here, the first device is buddied with the second, as indicated by the dark triangles. These are values that, when combined, will cancel each other out: so, the dark “down” triangle could be 2.0, and the dark “up” could be 0.5. When multiplied together, they give 1. Similarly, the first device is paired with the third device. Every device has two “buddies,” where the number on that device has a counterpart on the other.

The data from a particular device, represented by a circle in [Figure 20-14](#), can then be combined with the random factors before being sent to the server.

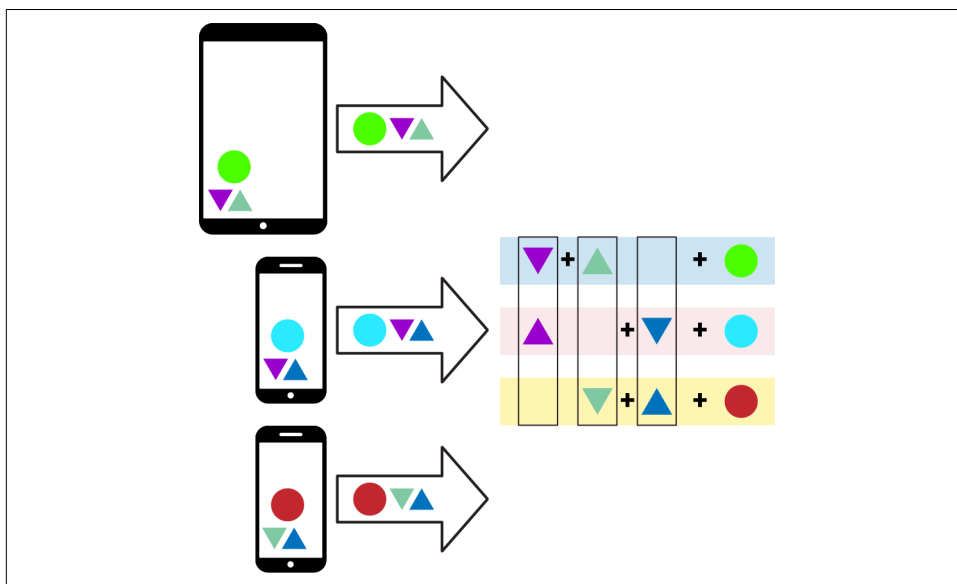


Figure 20-14. Sending values to the server with secure aggregation

The server, knowing the values sent to the buddies, can cancel them out and just get the payload. While the data is in transit to the server it is obfuscated by the keys.

Federated Learning with TensorFlow Federated

TensorFlow Federated (TFF) is an open source framework that gives you federated learning functionality in a simulated server environment. At the time of writing it’s still experimental, but it’s worth looking into. TFF is designed with two core APIs. The first is the Federated Learning API, which gives you a set of interfaces that add federated learning and evaluation capabilities to your existing models. It allows you to, for example, define distributed variables that are impacted by learned values from distributed clients. The second is the Federated Core API, which implements the federated communication operations within a functional programming environment. It’s the foundation for existing deployed scenarios such as the Google keyboard, [Gboard](#).

I won't go into detail on how to use TFF in this chapter because it's still in its early stages, but I encourage you to check it out to prepare for the day that on-device federated learning libraries become available!

Google's AI Principles

TensorFlow was created by Google's engineers as an outcropping of many existing projects built by the company for its products and internal systems. After it was open sourced, many new avenues for machine learning were discovered, and the pace of innovation in the fields of ML and AI is staggering. With this in mind, Google decided to put out a **public statement** outlining its principles with regard to how AI should be created and used. They're a great guideline for responsible adoption, and worth exploring. In summary, the principles are:

Be socially beneficial

Advances in AI are transformative, and, as that change happens, the goal is to take into account all social and economic factors, proceeding only where the overall likely benefits outstrip the foreseeable risks and downsides.

Avoid creating or reinforcing unfair bias

As discussed in this chapter, bias can easily creep into any system. AI—particularly in cases where it transforms industry—presents an opportunity to *remove* existing biases, as well as to ensure that *new* biases don't arise. One should be mindful of this.

Be built and tested for safety

Google continues to develop strong safety and security practices to avoid unintended harm from AI. This includes developing AI technologies in constrained environments and continually monitoring their operation after deployment.

Be accountable to people

The goal is to build AI systems that are subject to appropriate human direction and control. This means that appropriate opportunities for feedback, appeal, and relevant explanations must always be provided. Tooling to enable this will be a vital part of the ecosystem.

Incorporate privacy design principles

AI systems must incorporate safeguards that ensure adequate privacy and inform users of how their data will be used. Opportunities for notice and consent should be obvious.

Uphold high standards of scientific excellence

Technological innovation is at its best when it is done with scientific rigor and a commitment to open inquiry and collaboration. If AI is to help unlock knowledge in critical scientific domains, it should aspire to the high standards of scientific excellence that are expected in those areas.

Be made available for uses that accord with these principles

While this point might seem a little meta, it's important to reinforce that the principles don't stand alone, nor are they just for the people building systems. They're also intended to give guidelines for how the systems you build can be used. It's good to be mindful of how someone might use your systems in a way you didn't intend, and as such, good to have a set of principles for your users too!

Summary

And that brings us to the end of this book. It's been an amazing and fun journey for me to write it, and I hope you've found value in reading it! You've come a long way, from the "Hello World" of machine learning through building your first computer vision, natural language processing, and sequence modeling systems and more. You've practiced deploying models everywhere from on mobile devices to the web and the browser, and in this chapter we wrapped up with a glimpse into the bigger world of how and why you should use your models in a mindful and beneficial way. You saw how bias is a problem in computing, and potentially a huge one in AI, but also how there is an opportunity now, with the field in its infancy, to be at the forefront of eliminating it as much as possible. We explored some of the tooling available to help you in this task, and you got an introduction to federated learning and how it might be the future of mobile application development in particular.

Thank you so much for sharing this journey with me! I look forward to hearing your feedback and answering whatever questions I can.

A

accuracy

- for dropout-enabled LSTMs, 138
- impact of reduced learning rate with stacked LSTMs, 136
- improving with hyperparameter tuning, 183
- lower for test data than training data, 29
- with lower learning rate, 109
- for LSTM over 30 epochs, 133
- measuring prediction accuracy, 170
- reduced dense architecture results, 115
- reporting network accuracy, 28
- for stacked LSTM architecture, 135
- training accuracy versus validation accuracy, 108

activation functions

- definition of term, 26
- rectified linear unit (relu), 26, 193
- sigmoid activation function, 45
- softmax activation function, 26, 61

adam optimizer, 28

AI winter, xvi

Android apps

- creating TensorFlow Lite apps
 - step 1: create new Android project, 230
 - step 2: edit layout file, 232
 - step 3: add dependencies, 234
 - step 4: add TensorFlow Lite model, 236
 - step 5: write activity code for inference, 236

IDE for developing, 229

open source sample apps, 242

processing images

- defining output arrays, 241

reconciling image structure, 240

resizing, 240

running interpreter, 241

transforming images for neural networks, 239

TensorFlow Lite Android support library, 243

Android Studio, 229

antigrams, 86

arrayBuffer, 277

artificial intelligence (AI)

approach to learning, xvi

fairness in machine learning, 343

fairness in programming, 340-343

Google's AI principles, 354

online resources, xvii

prerequisites to learning, xvi

augmentimages function, 74

autocorrelation, 166

B

backpropagation, 106-107

batching, 175

BeautifulSoup library, 92, 97

bias

age bias, 345

education bias, 348

failure to consider, 340

identifying, 343

introducing, 347

learned by models, 17

obfuscating, 352

binary classification, 45

binary cross entropy loss function, 47

Brackets IDE, 265
ByteBuffer, 237, 277

C

callbacks, 31, 279
character-based encoding, 161
Civil Comments dataset, 295
cleaning text data, 91
code examples
 obtaining, xvii
 using, xviii
code generation, 242
comments and questions, xviii
computer vision neural networks
 avoiding overfitting, 30
 coding techniques in TensorFlow.js, 275-289
 complete code, 26-28
 accessing datasets, 27
 defining neural network, 28
 fitting training images to training labels, 27
 loss function and optimizer, 28
 model evaluation, 28
 network accuracy, 28
 normalizing images, 27
designing, 25-26
 activation functions, 26
 dense layer, 25
 hyperparameter tuning, 26
 input layer specification, 25
 output layer, 26
detecting features in images (see convolutional neural networks (CNNs))
exploring model output, 29
Fashion MNIST database, 22
neurons for vision, 23
recognizing clothing items, 21
stopping training, 30
training the neural network, 29
Conv1D layer, 193
Conv2D layer, 37
convolutional layers, 37
convolutional neural networks (CNNs)
 building in JavaScript, 277
 classifying contents of images, 42-52
 adding validation to Horses or Humans dataset, 47
 CNN architecture for Horses or Humans, 45

Horses or Humans dataset, 42
Keras ImageDataGenerator, 43
 testing horse or human images, 49
convolutions
 basics of, 34
 coding convolutions, 192
 for sequence data, 191
detecting features in images with, 33
dropout regularization, 63-65
exploring, 39-42
image augmentation, 52
implementing, 37-39
multiclass classification, 59-63
pooling, 35
transfer learning, 55-59

Core API, 264
CSV (comma-separated values) files, 97-98
csv library, 97
custom splits, 74

D

dataset.window, 175
datasets (see also Horses or Humans dataset; TensorFlow Data Services (TFDS))
 accessing built-in datasets in Keras, 27
 Civil Comments dataset, 295
 Dogs vs. Cats database, 58, 222
 Fashion MNIST database, 22, 71
 ImageNet dataset, 306
 Internet Movie Database (IMDb), 93-97
 KNMI Climate Explorer dataset, 203
 loading data splits, 69
 MNIST database, 22, 282
 News Headlines Dataset for Sarcasm Detection, 99, 103
 preprocessed datasets in TensorFlow, 8
 public, 67
 Sentiment Analysis in Text dataset, 97
 Stanford Question Answering Dataset (SQuAD), 99
 Swivel dataset, 123
 test data and training data, 29, 98, 101
 using custom splits, 74
 validation datasets, 48
 windowed datasets, 173-178
deep neural networks (DNNs)
 evaluating DNN results, 179
 flatten and dense layers, 39
 for sequence data prediction, 178

- dense layers, 15
- deployment
 - applying "what if" scenarios prior to, 346
 - benefits of TensorFlow for, 7
 - continual improvement following, 349
 - mobile devices, 328
 - monitoring operations following, 354
 - rapid redeployment, 344
 - server-based, 327
 - to mobile devices, 329
 - using TensorFlow Serving, 332
 - using TensorFlow.js, 263
- differencing, 171
- dimensionality, 192
- distribution strategy, 8
- Docker, 330
- Dogs vs. Cats database, 58, 222
- dropout regularization, 63-65, 116, 137, 206
- drop_remainder, 175
- dynamic range quantization, 225

E

- elastic regularization, 118
- Embedding Projector, 120
- embeddings
 - building a sarcasm detector, 106
 - embedding layers, 106
 - purpose of, 103
 - reducing overfitting in language models, 109-119
 - sentence classification, 119
 - using pretrained from TensorFlow Hub, 123
 - using pretrained with RNNs, 139-146
 - visualizing, 120-123
- estimators, 8
- ethics (see also bias; fairness; privacy)
 - building for, 340
 - Google's AI principles, 354
 - recent advances in ML and, 340
- expand_dims method, 51
- Extract-Transform-Load (ETL)
 - benefits of, 80
 - managing data in TensorFlow with, 78
 - optimizing the load phase, 80
 - parallelizing the ETL process, 81
 - phases of, 79

F

- Facets, 346

- fairness
 - in machine learning, 343
 - in programming, 340-343
 - tools for
 - Facets, 346
 - What-If Tool, 345
- Fashion MNIST database, 22, 71
- feature vectors, 319
- federated learning
 - goals of, 349
 - secure aggregation with, 352
 - steps of, 349-351
 - with TensorFlow Federated (TFF), 353
- filters, 34
- flatten layers, 39
- flat_map function, 175
- float16 quantization, 226
- Floating Point to Hex Converter , 293
- from_saved_model method, 219
- full integer quantization, 226

G

- gated recurrent units (GRUs) , 205
- Gboard, 353
- Goddard Institute for Space Studies (GISS) Surface Temperature analysis., 198
- Google Colab
 - model testing, 49
 - TFDS versions and, 73
 - using TensorFlow in, 12
- Google's AI principles, 354
- graphics processing units (GPUs), 80

H

- handwriting recognizer, 275
- hidden layers, 25
- Horses or Humans dataset
 - adding validation to, 47
 - CNN architecture for, 45
 - examples from, 42
 - testing horse or human images, 49
 - using TDFS with Keras models, 71
- HTML tags, 91, 97
- hyperparameter tuning
 - Conv1D layer, 195
 - definition of term, 26
 - with Keras Tuner, 185-188
 - options for, 160
 - other optimization considerations, 118

I

- image augmentation
 - benefits of, 52
 - using ImageDataGenerator, 53
 - other transformations, 53
 - using mapping functions for, 73
- image feature vector model types, 319
- ImageDataGenerator, 43, 53
- ImageNet dataset, 306
- images
 - assigning labels to automatically, 43
 - categorizing (see computer vision neural networks)
 - detecting features in (see convolutional neural networks (CNNs))
- inference, 7, 17
- input sequences, 148-152
- Internet Movie Database (IMDb), 93-97
- iOS apps
 - creating TensorFlow Lite apps
 - step 1: create basic iOS app, 245
 - step 2: add dependencies, 247
 - step 3: create user interface, 248
 - step 4: add and initialize model inference class, 250
 - step 5: perform the inference, 253
 - step 6: add model to app, 254
 - step 7: add UI logic, 256
 - IDE for developing, 245
- image processing
 - defining output arrays, 260
 - full app code, 258
 - invoking interpreter, 260
 - reconciling image structure, 259
 - resizing, 259
 - transforming images for neural networks, 258
- open source sample apps, 261

J

- JavaScript models
 - considerations for TensorFlow, 276
 - converting Python-based models to, 291-293
 - transfer learning
 - from MobileNet, 306-319
 - from TensorFlow Hub, 319-322
 - from TensorFlow.org, 322
 - overview of, 305

- using converted models, 293
- using preconverted models, 295-304
 - Toxicity Classifier, 295
 - using MobileNet for image classification, 298
 - using PoseNet, 301
- JavaScript Object Notation (JSON) files, getting text from, 99-101
- json library, 100

K

- Keras
 - accessing built-in datasets, 27
 - fitting training images to training labels, 47
 - ImageDataGenerator, 43
 - implementing dropouts, 64
 - ML basic code, 14
 - preprocessing library, 86
 - purpose of, 8
 - using TensorFlow Data Services with, 70-73
- Keras H5 format, 218
- Keras Tuner, 185-188
- kernel_regularizer property, 118
- KNMI Climate Explorer dataset, 203
- Kotlin, 229

L

- L1 (lasso) regularization, 117
- L2 (ridge) regularization, 118
- labels, 151, 174
- lambda functions, 175
- lasso regularization, 117
- latency, 216
- layers
 - Conv1D layer, 193
 - Conv2D layer, 37
 - convolutional layers, 37
 - defining, 15
 - dense layers, 15
 - diagram of, 14
 - embedding layers, 106
 - flatten layer, 39
 - hidden layers, 25
 - pooling layers, 38
 - SimpleRNN layer, 201
- Layers API, 264
- learning process, 16
- learning rate (lr) parameter, 47, 109, 136, 183
- Linux, 331

- load_data method, 27
 - loss curves
 - for dropout-enabled LSTMs, 138
 - impact of reduced learning rate with stacked LSTMs, 137
 - improving with hyperparameter tuning, 183
 - with lower learning rate, 110
 - with LSTM over 30 epochs, 134
 - reduced dense architecture results, 115
 - for stacked LSTM architecture, 136
 - training loss versus validation loss, 108
 - loss functions
 - binary cross entropy loss function, 47
 - mean squared error (MSE), 178
 - purpose of, 15
 - selecting, 28
 - sparse categorical cross entropy, 28
 - low latency, 216
 - LSTM (long short-term memory)
 - for sequence data, 205
 - high-level diagram, 130
 - optimizing stacked LSTMs, 136
 - purpose of, 130
 - stacking LSTMs, 134
 - text generation using, 152-161
 - using dropout, 137
- ## M
- machine learning (ML)
 - approach to learning, xvi
 - basics of
 - Keras code line-by-line, 14
 - layers, 14
 - learning process, 16
 - loss functions, 15
 - machine learning paradigm, 13
 - neural networks, 14
 - optimizers, 15
 - prediction, 17
 - weight and bias, 17
 - defined, 6
 - evolution from traditional programming, 1, 3, 5, 339
 - methodology at heart of, 339
 - microcontrollers and, 227
 - online resources, xvii
 - pipeline for, 328
 - prerequisites to learning, xvi
 - system architecture modules for, 327
 - training process, 7
 - MAJOR.MINOR.PATCH numbering system, 73
 - mapping functions, 73
 - max_sequence_len, 159
 - mean absolute error (MAE), 170
 - mean squared error (MSE), 170, 178
 - metadata, 242
 - microcontrollers, 227
 - mobile applications (see Android apps; iOS apps; MobileNet library; TensorFlow Lite)
 - MobileNet library
 - image classification with, 298
 - transfer learning from
 - overview of, 306
 - step 1: download MobileNet and identify layers, 306
 - step 2: create model architecture, 308
 - step 3: gather and format data, 310-315
 - step 4: model training, 316
 - step 5: run inference, 317
 - model.evaluate command, 29
 - model.fit command, 16
 - model.predict, 154
 - model.summary command, 39
 - models
 - creating, 7
 - efficient for mobile applications, 216
 - image feature vector model types, 319
 - predictive, 148
 - testing with Google Colab, 49
 - training process, 7
 - versioning, 329
 - ModelServerConfig, 335
 - Modified National Institute of Standards and Technology (MNIST) database, 22, 282
 - moving average, 170
 - multiclass classification, 59-63
 - multivariate time series, 164
- ## N
- naive prediction, 167
 - NASA Goddard Institute for Space Studies (GISS) Surface Temperature analysis, 198
 - natural language processing (NLP)
 - definition of term, 85
 - encoding language into numbers
 - approaches to, 85
 - tokenization, 86

- turning sentences into sequences, 87-91
- reducing overfitting in language models, 109-119
- removing stopwords and cleaning text, 91
- working with data sources
 - CSV files, 97-98
 - JSON files, 99-101
 - TensorFlow Data Services (TFDS), 93-97
- neural networks
 - computer vision, 21-32
 - convolutional, 33-66
 - deep neural networks (DNNs), 39
 - diagram of, 14
- Neural Networks API, 217
- neurons
 - in computer vision neural networks, 23
 - diagram of, 14
- News Headlines Dataset for Sarcasm Detection, 99, 103
- next_words parameter, 155
- noise, 166
- normalization, 27
- np.argmax, 154
- Numpy library
 - benefits of, 16
 - expand_dims method, 51

O

- one-hot encoding, 151
- online resources, xvii
- optimizations property, 225
- optimizers
 - adam optimizer, 28
 - purpose of, 15
 - root mean square propagation (RMSPProp), 47
 - selecting, 28
 - stochastic gradient descent (sgd), 178
- out-of-vocabulary (OOV) tokens, 88
- overfitting
 - avoiding, 30
 - causes of, 109
 - definition of term, 25
 - indications of, 108
 - overcoming with dropout regularization, 63-65
 - reducing in language models
 - adjusting learning rates, 109
 - exploring embedding dimensions, 113

- exploring model architecture, 115
- exploring vocabulary size, 111-113
- other optimization considerations, 118
- using dropout, 116
- using regularization, 117

P

- padding, 89-91
- pad_sequences API, 90
- parallelization, 81
- parameters, 24
- pooling, 35, 38
- PoseNet library, 301
- power consumption, 216
- prediction
 - measuring prediction accuracy, 170
 - predicting text, 147
 - use of term, 17
- preprocessing library, 86
- prerequisites, xvi
- privacy
 - maintaining user's, 349-351
 - secure aggregation and, 352
- protobuf format, 335
- punctuation, removing, 92
- PyCharm, 9
- Python
 - converting Python models to JavaScript, 291-295
 - csv library, 97
 - json library, 100
 - reading GISS data in, 199
 - standalone interpreter for, 221
 - string library, 92

Q

- quantization, 225
- questions and comments, xviii

R

- ragged tensors, 91
- rapid redeployment, 344
- Raspberry Pi, 221
- rectified linear unit (relu), 26, 193
- recurrence, 127
- recurrent dropout parameter, 206
- recurrent neural networks (RNNs)
 - basis of recurrence, 127

- bidirectional, 209
- creating text classifiers, 132
- extending recurrence for language, 130
- optimizing stacked LSTMs, 136
- stacking LSTMs, 134
- using dropout, 137
- using for sequence modeling, 200-205
- using pretrained embeddings with, 139-146
- regularization, 117
- return_sequences property, 134, 157
- ridge regression, 118
- root mean square propagation (RMSProp) optimizer, 47

S

- SavedModel format, 218
- seasonality, 165
- secure aggregation, 352
- seeds, 147, 154
- sentence classification, 119, 127, 132
- sentence length, 118
- sentiment analysis
 - deriving meaning from numerics, 104
 - embeddings in TensorFlow
 - building a sarcasm detector, 106
 - embedding layers, 106
 - reducing overfitting in language models, 109-119
 - increasing dimensionality of word direction, 105
 - order of words and, 127
 - RNNs for
 - basis of recurrence, 127
 - creating text classifiers, 132
 - extending recurrence for language, 130
 - optimizing stacked LSTMs, 136
 - stacking LSTMs, 134
 - using dropout, 137
 - using pretrained embeddings with, 139-146
- Sentiment Analysis in Text dataset, 97
- sequence prediction
 - convolutions for
 - coding convolutions, 192
 - experimenting with hyperparameters, 195
 - one-dimensional, 191
 - creating and training DNNs to fit, 178
 - creating windowed datasets, 174

- creating windowed version of time series dataset, 176
- evaluating DNN results, 179
- exploring overall prediction, 181
- features and labels, 173
- hyperparameter tuning with Keras Tuner, 185-188
- tuning learning rate, 183
- using bidirectional RNNs, 209
- using dropout, 206
- using NASA weather data
 - reading GISS data in Python, 199
 - selecting weather station, 198
- using other recurrent methods, 205
- using RNNs for, 200-205
- shuffling, 175
- sigmoid activation function, 45
- SimpleRNN layer, 201
- softmax activation function, 26, 61
- sparse categorical cross entropy loss function, 28
- split parameter, 75
- sprite sheets, 282
- standalone interpreter, 221
- Stanford Question Answering Dataset (SQuAD), 99
- stochastic gradient descent (sgd), 15, 178
- stopwords, 91
- string library, 92
- string.punctuation constant, 92
- subwords, 95
- Swivel dataset, 123

T

- tensor processing units (TPUs), 8, 80
- TensorFlow
 - APIs for serving, 8
 - approaches to model training, 7
 - basics of, 7
 - creation of, 354
 - defining layers in, 15
 - ecosystem, 263
 - examining weights and biases, 18
 - high-level architecture of, 7
 - implementing dropouts in, 64
 - installing
 - in Python, 9
 - using in Google Colab, 12
 - using in PyCharm, 9

- preprocessed datasets, 8
- TensorFlow Addons library, 74
- TensorFlow Data Services (TFDS)
 - benefits of, 8
 - categories of available datasets, 67
 - getting started with
 - accessing datasets with, 68
 - examining datasets, 69
 - inspecting data type returned, 69
 - installing, 68
 - loading data splits in datasets, 69
 - viewing data about the dataset, 69
 - goal behind, 67
 - loading specific versions, 73
 - managing data with ETL process, 78-83
 - TFRecord structure, 76-78
 - using custom splits, 74
 - using mapping functions for augmentation, 73
 - using TensorFlow Addons library, 74
 - using with Keras models, 70-73
- TensorFlow Extended (TFX), 327
- TensorFlow Federated (TFF), 353
- TensorFlow Hub
 - lists of models optimized for TensorFlow Lite, 217
 - mobilenet_v2 model from, 222
 - transfer learning from, 319-322
 - using pretrained embeddings from, 123
- TensorFlow Lite (see also Android Apps; iOS apps)
 - constraints addressed by, 216
 - creating an image classifier
 - step 1: build and save the model, 222
 - step 2: converting the model, 223
 - step 3: optimizing the model, 225
 - creating and converting a model to
 - overview of, 217
 - step 1: saving the model, 218
 - step 2: converting and saving the model, 219
 - step 3: loading TFLite model and allocating tensors, 219
 - step 4: performing the prediction, 220
 - goals of, 215
 - history of, 215
 - main components of, 217
 - models optimized to work with, 217
 - tools provided by, 8
- TensorFlow Lite for Microcontrollers, 227
- TensorFlow Lite Micro (TFLM), 8
- TensorFlow Serving
 - benefits of, 327, 329
 - building and serving a model, 332-335
 - exploring server configuration, 335-337
 - installing
 - directly on Linux, 331
 - server architectures, 330
 - using Docker, 330
 - role in TensorFlow ecosystem, 328
- TensorFlow.js
 - basic model building, 266-270
 - creating image-based classifiers
 - building CNNs in JavaScript, 277
 - implementation details, 276
 - JavaScript considerations, 276
 - running inference on images, 288
 - training with MNIST dataset, 282-287
 - using callbacks for visualization, 279
 - creating iris classifier, 270-274
 - high-level architecture, 264
 - installing and using Brackets IDE, 265
 - purpose of, 263
 - role in TensorFlow ecosystem, 263
 - tools provided by, 8
 - transfer learning
 - from MobileNet, 306-319
 - from TensorFlow Hub, 319
 - overview of, 305
 - using models from TensorFlow.org, 322
 - using Python-created models with
 - converting models to JavaScript, 291
 - overview of, 291
 - using converted models, 293
 - using preconverted models, 295-304
- TensorFlow.org, 322
- tensorflowjs tools, 291
- text classification, 132
- text creation
 - changing model architecture, 157
 - character-based encoding, 161
 - extending the dataset, 156
 - generating text
 - compounding predictions, 155
 - predicting the next word, 154
 - improving the data, 158
 - model creation, 152
 - overview of, 147

- turning sequences into input sequences, 148-152
- text_to_sequences method, 87, 150, 154
- tf.browser APIs, 277
- tf.browser.fromPixels method, 277
- tf.browser.toPixels method, 277
- tf.Data.Dataset.list_files, 82
- tf.expand_dims statement, 192
- tf.image library, 74
- tf.keras.datasets, 27
- tf.keras.layers.Bidirectional, 209
- tf.keras.layers.Conv2D, 37
- tf.tidy, 287
- tfd.load, 68, 71
- tfjs-vis library, 280
- TFRecord structure, 76-78
- tidy API, 277
- time series data
 - baseline measurement, 167
 - basics of, 163
 - common attributes of
 - autocorrelation, 166
 - noise, 166
 - seasonality, 165
 - trend, 165
 - multivariate and univariate, 164
 - predicting
 - improving moving average, 171
 - measuring prediction accuracy, 170
 - naive prediction, 167
 - using moving average, 170
 - synthetic time series creation, 167
 - windowed version of, 176
- TinyML, 8, 227
- tokenization, 86, 111, 123
- Toxicity classifier, 295
- to_categorical utility, 152

- traditional programming, limitations of, 3
- training
 - definition of term, 7
 - improving performance with parallelization, 81
 - stopping training, 30
- transfer learning
 - idea behind, 55
 - in JavaScript
 - from MobileNet, 306-319
 - from TensorFlow Hub, 319
 - overview of, 305
 - using models from TensorFlow.org, 322
 - training horse-or-human classifier, 57
 - using prelearned layers, 55
- transformations, 53, 73
- trend, 165

U

- univariate time series, 164
- user privacy, 216

V

- validation datasets, 48
- versioning, 329
- visors, 279
- visualizations, 120-123, 279
- vocabulary size, 111

W

- weights, 17, 34
- What-If Tool, 345
- windowed datasets, 173-178
- windowing, 158
- with_info parameter, 69
- word_counts property, 111

About the Author

Laurence Moroney leads AI Advocacy at Google. His goal is to educate software developers in building AI systems with machine learning. He's a frequent contributor to the TensorFlow YouTube channel, youtube.com/tensorflow, a recognized global keynote speaker, and author of more books than he can count, including several best-selling science fiction novels, and a produced screenplay. He's based in Sammamish, Washington where he drinks way too much coffee.

Colophon

The animal on the cover of *AI and Machine Learning for Coders* is a lobe-footed or web-footed gecko (*Palmatogecko rangei*). There are over a thousand species of gecko, which are small lizards. This species evolved in the Namib Desert in Namibia and southwestern Africa.

Web-footed geckos are nearly translucent, four- to six-inch long creatures, with oversized eyes that they lick to keep clear. Their webbed feet move like scoops through the desert sand, and they have adhesive pads on their toes that further enhance their agility. Like most geckos, the web-footed variety are nocturnal insectivores.

Unlike other reptiles, most geckos can vocalize. The web-footed geckos have an especially broad range of vocalizations, including clicks, croaks, barks, and squeaks.

The web-footed gecko population has not been evaluated by the International Union for Conservation of Nature. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The color illustration is by Karen Montgomery, based on a black and white engraving from *Lydekker's Royal Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.